Escola Tècnica Superior d'Enginyeries
Industrial i Aeronàutica de Terrassa
UNIVERSITAT POLITÈCNICA DE CATALUNYA

GRAU EN ENGINYERIA EN TECNOLOGIES INDUSTRIALS

TREBALL FINAL DE GRAU

Estudi d'un gestionador de l'accessibilitat de pàrquings amb Raspberry Pi

MEMÒRIA

12/06/2015

Autor: Adrià Vidal Aguilar

Director: Jordi Saludes Closa

Contingut

Índex de figures

1 Plantejament del problema

1.1 Objecte

En aquest estudi es pretén crear un algoritme informàtic basat en el llenguatge de programació Python per a controlar un pàrquing exterior sense marcar. L'objectiu és aconseguir mitjançant visió artificial determinar quan un vehicle entra dins l'espai delimitat d'un pàrquing exterior i marcar la posició exacta on ha estacionat. Aconseguint segmentar o marcar de forma virtual l'extensió de terreny utilitzable per aparcar i donant pas a possibles anàlisis posteriors d'interès, com saber si un vehicle estacionat podria impedir el moviment d'altres.

1.2 Justificació

Els pàrquings exteriors tenen grans avantatges. Pràcticament qualsevol espai obert es pot utilitzar directament com a zona d'estacionament sense necessitat d'una enorme inversió econòmica, com en el cas dels pàrquings subterranis. Però, per altra banda, tenen un gran inconvenient, les places per aparcar no estan delimitades o marcades ja que el terreny és verge, no ha estat asfaltat ni cementat. Un pàrquing amb les places marcades dona lloc a un millor control sobre aquest, podent saber exactament on aparcarà cada vehicle, quants n'hi han i que aquests no impediran mai el moviment dels altres.
Per tant, la justificació de l'estudi és que mitjançant la utilització de hardware de molt baix cost i d'un senzill programa, es pugui controlar gràcies a la visió artificial on està aparcat cada vehicle en qualsevol mena de pàrquing exterior, creant així una segmentació virtual del possible espai d'estacionament. Així com permetre controlar remotament quan un usuari ha estacionat malament.

1.3 Abast

L'abast de l'estudi és crear un algoritme que:

- Una vegada iniciat el programa, capturi un *frame* rere un altre del terreny o superfície utilitzable del pàrquing a analitzar. Sent un *frame* una imatge en particular d'una successió d'imatges que formen una animació.
- Transformi la perspectiva de cada *frame* com si es capturessin des d'una vista aèria, ja que es capturaran amb un cert angle sobre la superfície del pàrquing.
- Analitzi mitjançant llibreries de visió artificial quan un vehicle entra a l'espai o pàrquing en qüestió i determinar la posició exacte on ha estacionat, creant una segmentació virtual per a cada vehicle sobre el terreny.

- Funcioni perfectament dins l'entorn de la computadora Raspberry Pi, la qual processarà tots els aspectes anteriors.

L'abast d'aquest estudi no inclou possibles anàlisis posteriors de les dades enregistrades per l'algoritme. La posició i mida enregistrada de cada vehicle sobre la superfície del pàrquing es podria utilitzar per a simular moviments dels vehicles dins el pàrquing, mitjançant eines de *motion planning*. Gràcies a aquesta simulació es podria veure si algun vehicle estacionat podria impedir el moviment d'altres, podent així evitar col·lapses dins el pàrquing.

1.4 Requeriments

Per al desenvolupament de l'estudi s'utilitzarà com a *hardware* la computadora Raspberry Pi i la càmera Raspberry Pi Camera Module.
Serà necessari també un terminal per a executar codi en llenguatge Python, que podrà ser executat des de la mateixa computadora Raspberry Pi.
Per a crear l'algoritme, s'utilitzarà la llibreria de funcions d'anàlisi de visió artificial OpenCV.
Finalment caldrà tindre accés visual a un pàrquing exterior sense marcar o crear una representació artificial d'aquest a petita escala.

2 Antecedents i estat de l'art

En les primeres fases de l'era dels ordinadors, era pràcticament impossible pensar en la visió artificial. S'ha d'entendre la visió artificial com aconseguir que una computadora entengui o analitzi una escena o les característiques d'una imatge. Els primers ordinadors, al disposar de processadors poc potents, eren incapaços de processar tota la informació que s'emmagatzema en una escena.
Posteriorment, cap al 1990, amb l'arribada de millores tecnològiques significatives en el rendiment dels ordinadors, així com la posterior creació de sensors d'imatge CMOS de baix cost que poden integrar funcions de processament de fotons directament en el circuit integrat, es va començar a investigar el camp de la visió artificial.
L'inici va ser marcat per Larry Roberts, qui va crear un programa que podia veure una estructura de blocs, analitzar el seu contingut i reproduir-la des d'una altra perspectiva, demostrant així que a informació visual havia estat enviada a l'ordinador mitjançant una càmera i aquest havia processat correctament. Posteriorment van anar sorgint tota mena de programes per a implementar la visió artificial en moltes àrees. (Luciano, 2013)

Tot i això, les funcions i llibreries que es creaven eren privades, i utilitzades només per a aplicacions particulars en la indústria en forma de programes complexos i d'alt cost.
El 1999 tot va canviar, l'empresa Intel, de circuits integrats, va desenvolupar una biblioteca lliure de visió artificial anomenada OpenCV. Gràcies a OpenCV, qualsevol persona pot accedir a centenars de funcions per a analitzar escenes de forma gratuïta. No només això, sinó que la biblioteca és multi plataforma, podent-se utilitzar en plataformes Linux, Windows o MacOS. OpenCV pretén proporcionar un entorn de desenvolupament fàcil d'utilitzar i altament eficient (OpenCV, 2012).

Actualment existeixen algoritmes creats per particulars amb tota mena de finalitats dins la visió artificial, des del reconeixement de cares al seguiment d'objectes en una escena.
Tanmateix existeixen també intents d'algoritmes per a monitorar pàrquings exteriors, però cap programa patentat i utilitzat en pàrquings reals.

3 Elements utilitzats en l'estudi

3.1 Raspberry Pi

El principal hardware utilitzat en aquest estudi és el Raspberry Pi. Es tracta d'un ordinador de placa reduïda de baix cost desenvolupat al Regne Unit. L'objectiu inicial del hardware era el d'estimular l'ensenyança de ciències de la computació a les escoles.

Figura 1: Placa Raspberry Pi

En l'estudi, el Raspberry Pi és utilitzat com a computadora per a analitzar cada *frame* capturat per la càmera i processar l'algoritme que permetrà la segmentació virtual del pàrquing.
El baix cost d'ell, al voltant de 35$, la seva reduïda mida, 85,60x56,5 mm, i el baix consum energètic, el fan perfecte per a poder-lo col·locar, per exemple, en una zona elevada amb una bona visibilitat del terreny d'un pàrquing, deixant-lo anar processant l'algoritme de segmentació durant hores.

Especificacions tècniques

El Raspberry Pi munta com a *System-on-a-chip* un Broadcom BCM2835. Un *SoC* (*System-on-a-chip*) és una tecnologia de fabricació que inclou tots o gran part dels mòduls components d'un ordinador o sistema informàtic en un circuit integrat o chip. El *SoC* utilitzat inclou CPU o unitat central de processament, GPU o unitat de processament gràfic, DSP o processament digital de senyals, SDRAM o memòries dinàmiques d'accés aleatori amb interfase síncrona y port USB o tecnologia de Bus Universal en Sèrie.

La CPU de la placa és una ARM 1176JZF-S a 700 MHz.
Com a GPU inclou una Breadcom VideoCore IV amb la llibreria OpenGL ES 2.0 i codificacions per a MPEG-2 i VC-1 amb llicència i 1080p30 H.264/MPEG-4 AVC.
La memòria (SDRAM) de la placa és de 512 MB compartits amb la GPU.

El Raspberry Pi disposa, també, de diversos ports externs: 4 ports USB 2.0; un connector que permet instal·lar un mòdul de càmera desenvolupat per la Fundació Raspberry Pi, com per exemple la Pi Camera, la qual s'utilitzarà també en aquest estudi; una sortida de vídeo amb connector RCA, HDMI i interfase per a panels LCD; una sortida d'àudio amb connector de 3,5 mm; i un connector Ethernet.
La unitat d'emmagatzemat integrat és mitjançant una unitat de memòria tipus MicroSD.

El consum energètic de la placa és de 600mA. Utilitza una font d'alimentació de 5V via Micro USB.

Els sistemes operatius suportats per la placa es basen en la plataforma Linux. S'hi pot instal·lar Debian (Raspbian), Fedora (Pidora) o Arch Linux (Arch Linux ARM). No obstant, la nova versió de Raspberry Pi, pretén suportar, també, el nou sistema operatiu Windows 10 de Microsoft.

(Foundation Raspberry Pi, 2008); (Upton & Halfacree, 2012)

3.2 Raspberry Pi Camera module

La Pi Camera es pot utilitzar per a capturar vídeo en alta definició fins a 1080p, així com *frames* del mateix, podent realitzar time-lapse. Aquesta es connecta directament a un port especial que disposa la placa Raspberry Pi.

Figura 2: Raspberry Pi Camera

Aquesta càmera s'utilitza en l'estudi per a capturar imatges en temps real de l'escenari a analitzar, en el nostre cas un pàrquing exterior.
La reduïda mida de la càmera, 25 × 20 × 9 mm, fan que, juntament amb l'ordinador Raspberry Pi, sigui de fàcil col·locació en qualsevol espai. A més, té un preu de 30$.

Especificacions tècniques

El mòdul té una càmera de focus fix de 5 megapíxels que suporta 1080p a 30fps (*frames* per segon), 720p a 60fps i vídeo VGA90, així com capturar imatges fixes a 5MP.

(Foundation Raspberry Pi, 2008)

3.3 Llenguatge Python

Python és un llenguatge de programació creat per Guido van Rossum a principis dels anys 90. Pretén ser un llenguatge amb una sintaxi molt neta i que afavoreixi un codi llegible.

S'ha decidit utilitzar Python per a la creació de l'algoritme de l'estudi ja que la seva sintaxi és simple, clara i senzilla. Per altre banda la gran quantitat de llibreries disponibles per a aquest llenguatge, com OpenCV, propicien a poder desenvolupar l'algoritme de forma més ràpida i senzilla que si es desenvolupés en altres llenguatges més complexos.

Python és un llenguatge interpretat o de script, de tipus dinàmic, tipus fort, multi plataforma i orientat a objectes.

Llenguatge interpretat o de script

És aquell llenguatge que s'executa utilitzant un programa entremig anomenat intèrpret, en comptes de compilar el codi a llenguatge màquina perquè la computadora pugui executar-lo directament. No es tracta, doncs, d'un llenguatge compilat, com si ho és, per exemple, el llenguatge C.

Els llenguatges compilats tenen l'avantatge de ser més ràpids a l'hora d'executar-se, però són menys flexibles y menys portables que els llenguatges interpretats.

No obstant, es podria dir que Python és un llenguatge semi interpretat, ja que el codi font es tradueix a un pseudo codi màquina anomenat *bytecode* la primera vegada que s'executa, generant arxius ".pyc", que són els que s'aniran executant successivament.

Tipus dinàmic

Significa que no és necessari declarar el tipus de dada que contindrà una determinada variable, sinó que el seu tipus es determinarà en el temps d'execució segons el tipus de valor que se li hagi assignat. A més, si

posteriorment se li assigna a la mateixa variable un valor d'un altre tipus, aquesta es canviarà pel nou.

Tipus fort

Significa que un cop inicialitzada una variable amb un tipus concret (nombre enter, decimal, booleà...) aquesta no es pot utilitzar com si fos d'un altre tipus dins un algoritme, en tot cas, cal convertir la variable al nou tipus desitjat, prèviament.

Multi plataforma

El programa intèrpret per executar codi Python està disponible per a tot tipus de plataformes, com Linux, Windows, MacOS, etc. Això significa que si no utilitzem llibreries específiques de cada plataforma, un programa creat amb Python podria funcionar en qualsevol d'aquests sistemes.

Python és del tot compatible amb el hardware utilitzat en l'estudi, el Raspberry Pi, ja que el sistema operatiu que munta està basat en Linux.

Orientat a objectes

L'orientació a objectes és un paradigma de programació en el que els conceptes del món real rellevants pel nostre problema es traslladen a classes i objectes en el nostre programa. L'execució del programa consisteix en una sèrie d'interaccions entre aquests objectes.

(González Duque, 2010)

3.4 Llibreria OpenCV

OpenCV és una llibreria de programació desenvolupada especialment per al processament d'imatges en temps real. Va ser creada el 1999 a l'empresa Intel per Gary Bradski, amb l'objectiu d'accelerar les aplicacions de visió per ordinador, de forma que cada vegada que es fes una nova aplicació, es pogués disposar d'algoritmes ja estudiats i optimitzats.

La primera versió alfa de OpenCV va aparèixer a la IEEE Conference on Computer Vision and Pattern Recognition l'any 2000. La primera versió oficial no beta va aparèixer el 2006.

Una de les raons per les quals OpenCV s'ha convertit en una de les llibreries de visió més conegudes, a part de la seva potència i gran varietat d'algoritmes de processament d'imatge, és gràcies a la seva publicació sota la llicència BSD, que permet que sigui usada lliurement per propòsits comercials i d'investigació de forma gratuïta. Entre la gran varietat d'algoritmes que disposa es troben eines

per, per exemple, detectar característiques 2D i 3D, segmentar i reconèixer objectes o seguir el moviment.

Open CV va ser escrita originalment en llenguatge C, tot i que a la versió del 2008 es va millorar substancialment la interfase amb llenguatge C++. A part, OpenCV també pot ser utilitzada en Java, Python, Ruby o C#. A més, es tracta d'una llibreria multiplataforma, per tant funciona en entorns Linux, Windows, MacOS, Android o iOS.

Per tots els avantatges anomenats anteriorment, per ser la llibreria de visió artificial amb més catàleg d'algoritmes i poder funcionar dins el Raspberry Pi en llenguatge Python, tot l'algoritme creat en l'estudi per a segmentar pàrquings utilitzarà tota mena de funcions extretes de la llibreria OpenCV.

(Bradski, 2008)

3.5 Llibreria NumPy

NumPy és una extensió de Python, que li agrega major suport per vectors i matrius, constituint una biblioteca de funcions matemàtiques d'alt nivell per operar amb aquests vectors o matrius. NumPy és una llibreria de codi obert i té múltiples desenvolupadors.
NumPy pretén crear operadors i funcions eficients per a treballar amb vectors amb una ràpida execució, com si es tractessin d'algoritmes usats en llenguatges compilats, com el C.

NumPy és usat en aquest estudi per emmagatzemar tota mena de dades dels *frames* que es capturen en forma de matrius. Podent comparar-les posteriorment per a buscar diferències en cada *frame*.

(Oliphant, 2006)

4 Possibles solucions i alternatives

Aquest estudi, al tenir un clar objectiu, que és aconseguir posicionar i segmentar virtualment un vehicle sobre un terreny, els passos a seguir a l'hora de desenvolupar l'algoritme són pràcticament invariables. Fent una ràpida pinzellada als passos a seguir, ja que més endavant es definirà en detall tot l'algoritme; el primer pas és inicialitzar les llibreries que s'utilitzaran, seguidament importar cada *frame* de la captura en vídeo. Després cal transformar la perspectiva del primer *frame*, que serà utilitzat com a *background* (o tapís), per a comparar qualsevol *frame* posterior amb aquest i analitzar si ha canviat alguna cosa de l'escenari, poden ser, per exemple, l'entrada d'un vehicle. Una vegada l'algoritme ha determinat que existeix un vehicle a l'escenari, aquest seguirà el seu moviment, comparant *frame* rere *frame* fins a determinar que el vehicle s'ha aturat. En aquell moment, l'algoritme en determinarà la posició final, l'emmagatzemarà en una llista i en dibuixarà un rectangle virtual al seu voltant, podent així saber, fins i tot, la mesura aproximada del vehicle sobre el terreny .

Havent explicat, de forma resumida, els passos que cal desenvolupar per a què l'algoritme funcioni correctament, l'única alternativa o decisió que s'ha de prendre en un principi, és decidir si l'algoritme s'escriurà usant la llibreria OpenCV sota l'entorn del llenguatge Python, o escriure'l utilitzant la famosa eina de software matemàtic Matlab.
A continuació s'explica quins avantatges i desavantatges té OpenCV sobre Matlab, i el perquè de la decisió final:

Facilitat d'ús

L'entorn Matlab, utilitza el seu propi llenguatge de programació anomenat M. És relativament fàcil d'utilitzar ja que és un llenguatge interpretat d'alt nivell. Això significa que l'usuari no s'ha de preocupar per temes com inicialitzar biblioteques, declarar variables, gestionar memòria o altres problemes que tenen llenguatges interpretats de baix nivell.

En canvi, en OpenCV, al utilitzar en el nostre cas Python, tot i ser un llenguatge de sintaxi clara i senzilla, és necessari un coneixement més alt de programació en un principi que en utilitzar Matlab.

Velocitat

En aquest aspecte el llenguatge M de Matlab és inferior a Python ja que aquest llenguatge es basa en Java, i Java es basa en C. Per tant, a l'hora d'anar a executar el programa, la maquina primer interpreta el codi, el converteix a Java i finalment l'executa.

En canvi, les funcions de OpenCV, al estar directament escrites en Python, es poden executar de forma molt més rapida. Ja que Python és un llenguatge

semi interpretat com s'ha explicat anteriorment. Un programa escrit en Python i usant llibreries OpenCV, s'executaran més ràpid i podran processar més informació en menys temps que el mateix programa escrit en Matlab. Per exemple, en qüestions de *frames,* un programa en Matlab pot analitzar uns 3-4 *frames* per segon, quan utilitzant la llibreria OpenCV es poden analitzar més de 30 *frames* per segon. Per tant, en termes de visió artificial en temps real, Matlab és poc eficient.

Recursos necessaris

L'algoritme creat en aquest estudi es basa en analitzar imatges i processar-les. La naturalesa d'aquesta acció és molt pesada en recursos ja que s'han de processar enormes quantitats de píxels en cada imatge. Consumir recursos en un ordinador es tradueix en la quantitat de RAM que cal utilitzar per a fer funcionar un programa en temps real.

En aquest aspecte Matlab consumeix una enorme quantitat de RAM en comparació amb un programa executat en codi Python.

Preu

La llicència per ús comercial de Matlab té un preu al voltant dels 1500€. En canvi, la llibreria OpenCV és totalment gratuïta i es pot utilitzar per a finalitats comercials.

Entorn de desenvolupament

Matlab disposa del seu propi entorn de desenvolupament. En canvi la biblioteca OpenCV, es pot utilitzar en tot tipus d'entorns, ja sigui C, Python, Ruby, etc. En el cas d'aquest estudi, s'utilitza el propi entorn de Python per a executar les funcions de OpenCV.

Portabilitat

En aquest aspecte els dos entorns estan en igualtat de condicions. Ja que és possible utilitzar-los tant en Linux, Windows o MacOS.

Adquisició d'habilitats de programació

En aquest aspecte, l'opció d'utilitzar la llibreria OpenCV, que té centenars de funcions basades tant en llenguatge C com en Python, és molt més profitosa a nivell acadèmic que escollir Matlab, on s'adquireixen coneixements en llenguatge M, molt menys utilitzat en el món informàtic que C o Python.

En conclusió, s'ha escollit l'opció d'utilitzar la llibreria OpenCV sota l'entorn del llenguatge Python degut al gran ventall de funcions que ofereix, la seva alta capacitat de processament, i les útils habilitats de programació que s'adquireixen durant la creació de l'algoritme.
(Utkarsh, 2012)

5 Algoritme

A continuació s'explica com funciona cada línia del codi de l'algoritme desenvolupat, així com les funcions utilitzades.
Davant de cada línia de codi apareix un claudàtor amb un número a dins, que indica en quina línia es troba dins l'algoritme.

A l'annex de l'estudi es troba el codi complet ordenat de l'algoritme.

L'estructura de l'algoritme és, per ordre d'aparició:

- Importar les llibreries que seran utilitzades.
- Inicialitzar la càmera.
- Petit algoritme per a que l'usuari decideixi la superfície útil de pàrquing.
- Creació de la funció per a canviar la perspectiva de la superfície útil.
- Creació de la funció que determina quan un vehicle entra o surt de l'escenari i en guarda les seves coordenades.
- Creació de la llista de vehicles registrats.
- Llaç principal que fa servir les funcions anomenades anteriorment per a afegir o eliminar cada vehicle de la llista de vehicles registrats, així com situar-lo virtualment sobre l'escenari.

5.1 Importar llibreries

El primer pas en tot algoritme que faci servir funcions de determinades llibreries és importar aquestes. En llenguatge Python s'utilitza la comanda **import** seguit del nom de la llibreria.

```
[1] import cv2
[2] import numpy as np
[3] import time
```

La primera llibreria a importar és l'OpenCV, ja explicada anteriorment. Aquesta llibreria permet utilitzar qualsevol de les seves funcions de visió artificial. S'importa com a "cv2" ja que en aquest algoritme s'utilitza la versió 2 de la llibreria OpenCV. S'ha decidit utilitzar aquesta versió, tot i existir la versió 3, ja que aquesta és més estable, la següent versió està encara en vies de desenvolupament i revisió.

Seguidament, s'importa la llibreria NumPy, també explicada anteriorment. Al costat de "import numpy" s'afegeix "as np", això significa que les funcions usades que utilitzin NumPy estaran referenciades amb el prefix "np."
Finalment cal importar una llibreria molt utilitzada que és la de Time. Aquesta permet utilitzar qualsevol referència temporal en l'algoritme, com per exemple marcar una certa quantitat de segons abans que s'inicialitzi una funció determinada.

5.2 Inicialització de la càmera

El següent pas després d'importar les llibreries és inicialitzar la càmera, o el que és el mateix, comunicar al algoritme que es farà servir una càmera determinada i s'importarà el que aquesta capturi.

[4] cam=cv2.VideoCapture(0)

Aquí ja s'utilitza la primera funció de la llibreria OpenCV. Cal remarcar que totes les funcions que pertanyin a aquesta llibreria portaran el prefix "cv2." davant.

La funció **cv2.VideoCapture()** únicament serveix per a llegir el que captura una determinada càmera connectada a la placa de l'ordinador utilitzat. Normalment, si no hi ha més d'una càmera connectada, s'introdueix el número zero entre els parèntesis, fent referència a que s'importarà vídeo de la primera càmera o càmera principal. En qualsevol ordinador aquesta serà la pròpia *webcam*.
Per a que la màquina entengui que s'ha de capturar vídeo, cal crear un objecte (referenciar una paraula a una funció determinada) "VideoCapture", per això s'introdueix la paraula "cam". Això significa que l'objecte "cam" capturarà vídeo, en aquest cas, de la càmera principal (Mordvintsev & Abid, 2014).

5.3 Petit algoritme de decisió de la superfície útil

Un cop inicialitzada la càmera, s'ha decidit crear un petit algoritme per a què l'usuari pugui interactuar amb el programa. Això consisteix en què a l'usuari se li mostrarà una imatge del pàrquing, capturada per exemple des d'un fanal situada a l'exterior de la superfície útil, on estarà situada la càmera. Llavors, l'usuari mitjançant el ratolí, haurà de triar els quatre punts que determinaran l'escenari on l'algoritme haurà de determinar l'entrada o sortida de vehicles. Aquests quatre punts, seran utilitzats després per a canviar la perspectiva dels frames que capturi la càmera, convertint cada frame en una imatge zenital del pàrquing, quan realment les imatges s'han capturat amb un cert angle d'inclinació respecte a la superfície.

[5] M=[]

Primer cal crear una llista buida, anomenada "M" on posteriorment s'introduiran les coordenades dels punts estriats per l'usuari.
Una llista és una seqüència d'espais on a cada un s'hi poden emmagatzemar des de vectors a simples caràcters o números.

```
[6] print ("Instruccions per a determinar la superfície del pàrquing:")
[7] print (" - Indica amb el ratolí els quatre punts que formaran la regió útil del pàrquing.")
[8] print (" - Segueix l'ordre: punt superior esquerra, superior dret, inferior esquerra i inferior dret.")
[9] print (" - Prem la tecla Esc al finalitzar.")
```

Les línies 6, 7, 8 i 9 de codi simplement mostren a l'usuari les instruccions per definir els quatre punts que delimiten la superfície del pàrquing. La funció **print()** de Python mostra per pantalla el que s'escriu entre cometes a continuació. (Python Foundation, 2015)

```
[10] print "En 10 segons es farà una captura del pàrquing.",
[11] for k in range (10):
[12]     print '.',
[13]     time.sleep(1)
[14] r,frame=cam.read()
```

A la línia 10 es mostra a l'usuari que en deu segons es realitzarà una captura de l'escenari.
A la línia 11 s'utilitza la comanda **for** per a crear un bucle. Aquest comando s'utilitza per a recórrer posicions, en aquest cas deu unitats, permetent efectuar accions o operacions a cada posició fins a recórrer per complet la funció adjacent al **for**. (Bartolomé Sintes, 2015)
Es crea una variable "k" on en un rang de 10 unitats ("in range(10)") s'anirà mostrant un punt ("print '.') cada segon. La funció **time.sleep()** efectua una pausa dels segons que s'indiquin dins el parèntesi abans de prosseguir. Ajuntant tot 'anomenat anteriorment, s'aconsegueix mostrar deu punts per pantalla, un cada segon, que simbolitzen els deu segons d'espera abans que es realitzi la captura (Python Foundation, 2015).

```
[14] r,frame=cam.read()
```

Finalment a la línia 14 s'utilitza la funció **.read()**. Aquesta s'utilitza per a llegir l'objecte que se li fica al davant del punt, en aquest cas "cam". Per tant, aquesta funció retornarà la lectura d'un *frame* del vídeo que s'estigui capturen. A més a més també retornar un booleà (cert o fals) per indicar si la lectura ha estat correcte o no. Al retornar dues coses, cal igualar la funció a dues variables, en aquest cas anomenades "r" (emmagatzemarà el booleà) i "frame" (emmagatzemarà el *frame* de l'instant en què es llegeixi la captura de vídeo). Cal recordar que un *frame* és cada una de les imatges, que en simultaneïtat, conformen un vídeo (Mordvintsev & Abid, 2014).

Figura 3: Representació d'un pàrquing buit

```
[15] def on_mouse(ev,x,y,flags,params):
[16]     if ev==cv2.EVENT_LBUTTONDOWN:
[17]             M.append([x,y])
[18] while 1:
[19]     cv2.namedWindow('pantalla')
[20]     cv2.imshow('pantalla',frame)
[21]     cv2.setMouseCallback('pantalla',on_mouse,0)
[22]     if cv2.waitKey()==27:
[23]             cv2.destroyAllWindows()
[24]             break
[25] print M
```

A la línia 18, s'introdueix el primer **while**. En llenguatge de programació la comanda **while** s'utilitza per a crear un bucle sempre que la condició imposada

per a què funcioni no es vegi alterada dins el mateix bucle. En aquest cas, el bucle finalitzaria, sortint d'aquest i permetent que es compilin les següents línies. Una altra forma de sortir del bucle és amb la comanda **break**, qui finalitzarà immediatament el llaç (Bartolomé Sintes, 2014).
En aquest algoritme, la condició imposada és el número 1 que es mostra al costat del **while**. Això significa que es vagi efectuant sempre el bucle, podent sortir d'aquest únicament quan es trobi amb un **break**.

[19] cv2.namedWindow('pantalla')

Dins del bucle, trobem, a la línia 19, la funció **cv2.namedWindow()**. Aquesta funció crea una finestra virtual que pot ser usada per allotjar, per exemple, una imatge. En aquest cas serà anomenada "pantalla" (OpenCV dev team, 2015).

[20] cv2.imshow('pantalla',frame)

A la línia 20 s'utilitza la funció **cv2.imshow('winname',image)**. Per a utilitzar-la cal definir primer el 'winname', que serà el nom de la finestra virtual que es crearà (en aquest cas s'ha escollit el nom de "pantalla"), i en segon lloc "imatge", que serà el nom de l'objecte que contindrà una imatge i es vol mostrar dins la finestra (en aquest cas el *frame* de la superfície del pàrquing) (OpenCV dev team, 2015).

[21] cv2.setMouseCallback('pantalla',on_mouse,0)

A la línia 21 s'utilitza la funció **cv2.setMouseCallback('windowName',onMouse,0)**. Aquesta funció crea un vincle entre una determinada finestra i una funció on hi intervingui algun botó del ratolí. Els dos paràmetres d'entrada de la funció ('windowName' i onMouse) són, respectivament, el nom de la finestra on s'analitzaran les accions efectuades pel ratolí, i el nom de la funció que captarà que s'ha premut algun botó del ratolí.
Aquesta funció està creada unes línies més amunt, concretament a la línia 15, ja que les funcions s'han de definir abans de ser utilitzades.

```
[15] def on_mouse(ev,x,y,flags,params):
[16]     if ev==cv2.EVENT_LBUTTONDOWN:
[17]             M.append([x,y])
```

En llenguatge de programació les funcions, també anomenades subrutines, es creen quan hi ha tasques que es volen poder utilitzar en diferents punts d'un programa. Concretament, en el llenguatge Python, per definir una funció es fa

servir la comanda **def** seguit del nom escollit per a definir la funció. Al costat del nom, entre parèntesis, cal definir quins arguments (o variables) cal entrar-li o definir per a què aquesta funcioni.

En el cas de la funció utilitzada, "on_mouse(ev,x,y,flags,params)", només cal fixar-se en els tres primers arguments d'entrada. El primer "ev" (event), com es mostra a la línia 16, és l'objecte creat per a igualar i utilitzar la funció d'OpenCV **cv2.EVENT_LBUTTONDOWN**. Aquesta funció únicament reconeix les coordenades del punt on s'ha premut el botó esquerre del ratolí. Aquestes coordenades quedaran emmagatzemades com a "x" i "y" (Shermal, 2013).

La funció està determinada a l'algoritme com a la condició d'un **if**. En el llenguatge de programació, el **if** s'utilitza per què la màquina només executi les accions descrites dins el **if** sota una condició donada. Si aquesta condició no es compleix, simplement la maquina passarà a executar les línies de codi posteriors a les que es situen dins del **if**. Fàcilment reconegudes ja que estan sagnades cap a la dreta (Bartolomé Sintes, 2015).

En aquest cas, cada vegada que es detecti una premuda del botó esquerre del ratolí, l'algoritme podrà passar a executar la següent línia de codi.

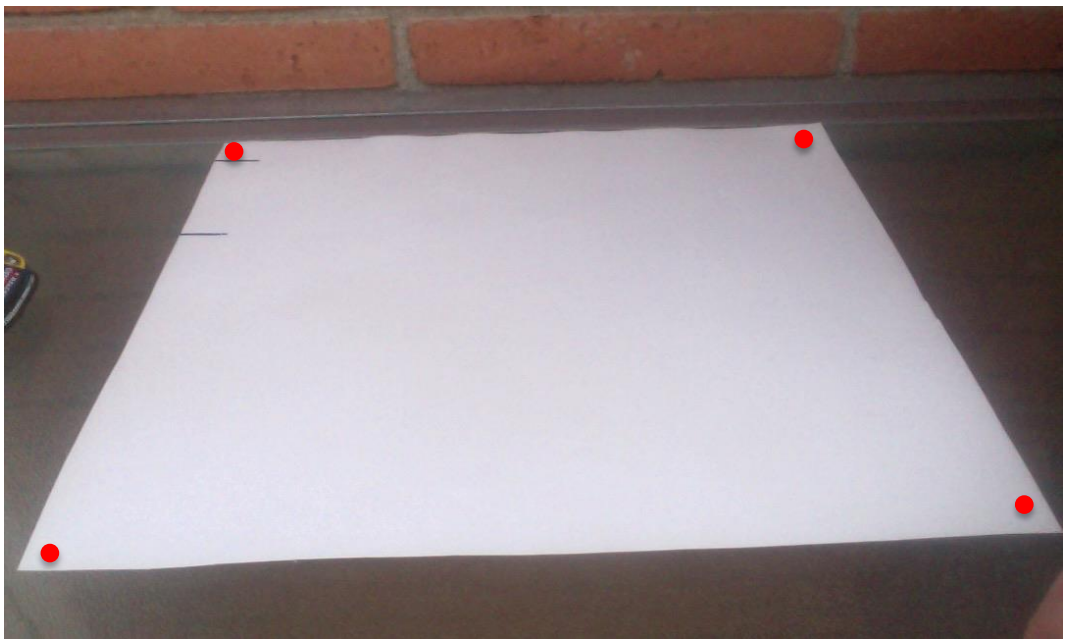

Figura 4: Coordenades que l'usuari hauria d'indicar

```
[17]            M.append([x,y])
```

La següent línia és la 17. En aquesta línia es pretén guardar cada coordenada sorgida al prémer el botó esquerre del ratolí sobre algun punt de la finestra on es mostra el *frame* de la superfície del pàrquing. Per a fer-ho, s'utilitza la funció **.append()**. Per a utilitzar-la, primer cal definir on vols afegir aquesta informació,

en aquest algoritme, les coordenades s'afegiran a la llista creada anteriorment, i fins ara buida, "M". Entre els parèntesis, cal definir les variables que s'afegiran, en aquest cas "x" i "y". El .**append()** sempre guardarà la informació a l'última posició de la llista. Una vegada l'usuari premi quatre punts del *frame* de la superfície, la llista "M" contindrà les quatre coordenades d'aquests punts (Python Foundation, 2015).

```
[22]    if cv2.waitKey()==27:
[23]            cv2.destroyAllWindows()
[24]            break
```

Finalment, les línies 22, 23 i 24 serveixen per a tancar la finestra, una vegada l'usuari ja hagi premut els quatre punts. La funció **cv2.waitKey()** acompanyada del **if** únicament serveix per a indicar que el sistema no entri a les tasques dins del **if** si no es prem una determinada tecla (OpenCV dev team, 2015). En aquest cas, la tecla que cal prémer és la número 27. Aquest número pertany a la coneguda tecla "Esc" (*escape*) (Binario, 2012). Per tant, quan l'usuari premi "Esc", la maquina entrarà i executarà la següent línia, la 23. La funció usada en aquesta línia és **cv2.destroyAllWindows()**. En executar-la es tanquen totes les finestres que s'hagin creat (OpenCV dev team, 2015).
Finalment s'executarà la comanda **break**, a la línia 24. Aquest comando s'utilitza per sortir automàticament del **while** introduït a la línia 18.

```
[25] print M
```

Una vegada explicat com l'algoritme mostra el *frame* de la superfície i en guardar les coordenades dels punts que la delimiten, finalment a la línia 25, gràcies al comando **print** es mostra quins valors ha emmagatzemat la llista "M". Això és de gran utilitat, ja que així l'usuari pot comprovar que efectivament ha marcat els 4 punts necessaris, i que les coordenades d'aquests són correctes.
A partir d'aquesta línia ja no hi ha més interacció entre el programa i l'usuari.

5.4 Funció per a canviar la perspectiva

Una vegada l'usuari hagi escollit els punts que delimitaran la superfície útil del pàrquing, aquests seran utilitzats per a transformar la perspectiva de cada *frame* que capturi la càmera. Transformant una superfície vista des d'un cert angle d'inclinació sobre el pla, i de forma trapezoïdal, en una superfície totalment rectangular i plana, fent que cada *frame* sembli capturat des d'una visió zenital. Aquesta transformació de la perspectiva permetrà agilitzar i facilitar el càlcul de

les diferències entre un *frame* i el posterior que determinaran i seguiran el moviment dels vehicles que entrin o surtin de l'escenari.

Aquesta funció s'ha anomenat "captura". Ja que inclou tant l'acció de capturar un nou *frame* com la de canviar la perspectiva d'aquest.

```
[26] def captura(cam,M):
[27]     r, frame = cam.read()
[28]     assert r
[29]     pts1=np.float32(M)
[30]     pts2=np.float32([[0,0],[1280,0],[0,720],[1280,720]])
[31]     transf=cv2.getPerspectiveTransform(pts1,pts2)
[32]     frame=cv2.warpPerspective(frame,transf,(1280,720))
[33]     return frame
```

Com es mostra a la línia 26 , es crea la funció anomenada "captura" mitjançant la comanda **def**. Els arguments d'entrada d'aquesta funció, o el que és el mateix, els elements que utilitzarà per a executar-se són l'objecte "cam", que fa referència a la lectura del que captura la càmera (explicat anteriorment) i la llista "M", que conté les coordenades dels punts que delimiten la superfície útil del pàrquing.

```
[27]     r, frame = cam.read()
```

La línia 27 funciona exactament igual que la ja explicada línia 14. Retornarà un *frame* guardat sota la variable anomenada "frame" i un booleà "r". Aquest serà el *frame* al qual se li aplicarà el canvi de perspectiva.

Figura 5: Frame qualsevol

[28] assert r

La comanda **assert** significa per assegurar que una variable existeix. En aquest cas, s'ha introduït un **assert** i la variable booleana "r". Això assegura que el *frame* s'ha capturat correctament i per tant, la "r" tindrà un valor booleà de *True*. Si existís qualsevol problema amb la càmera, i aquesta no captures correctament els *frames*, la variable booleana "r" seria *False* i no compliria el **assert**. Per tant el programa es pararia i donaria un error, permetent revisar el funcionament de la càmera o qualsevol altre aspecte que pugui haver propiciat una mala captura d'un *frame*.

[29] pts1=np.float32(M)
[30] pts2=np.float32([[0,0],[1280,0],[0,720],[1280,720]])
[31] transf=cv2.getPerspectiveTransform(pts1,pts2)

A la línia 31 es fa servir la funció **cv2.getPerspectiveTransform(src,dst)**, la qual fa una transformació de perspectiva entre quatre parells de coordenades, introduïts a la posició "src" i els converteix en els quatre parells de coordenades imposats a la posició "dst". El resultat és una matriu que emmagatzema aquesta informació (OpenCV dev team, 2015).
En el cas d'aquest algoritme, els primers quatre parells de coordenades s'han anomenat "pts1", a la línia 29. Per determinar aquests quatre punts s'utilitza la matriu "M" obtinguda en l'apartat anterior de l'algoritme. Però cal transformar aquesta llista en un una variable tipus *float32* mitjançant una de les funcions

disponibles a la llibreria NumPy (np), per a què la funció **cv2.getPerspectiveTransform()** pugui treballar. Les variables *float32* emmagatzemen números en forma de coma flotant i 32bits.
A la línia 30, es crea el segon paquet de quatre parells de coordenades. Després de transformar la perspectiva del *frame* aquest ocuparà una superfície rectangular on cada vèrtex estarà en les coordenades escollides a "pts2". En aquest cas, les coordenades formen un escenari de 1280 píxels d'amplada per 720 píxels. S'ha escollit aquesta mida ja que s'encabeix en una pantalla de 13 polsades i té una alta densitat de píxels. No obstant, degut a l'elecció d'aquesta mida, visualment l'àrea del pàrquing es veurà transformada, no serà proporcional a la realitat.
Aquesta transformació obtinguda, es guarda sota la variable anomenada "transf".

```
[32]    frame=cv2.warpPerspective(frame,transf,(1280,720))
```

Per últim, a la línia 32 s'utilitza la funció **cv2.warpPerspective()** la qual aplica la transformació d'unes coordenades a unes altres extretes de la funció **cv2.getPerspectiveTransform()** a una imatge escollida. En aquest cas la imatge escollida és el *frame* importat. Aquesta funció pren el *frame* importat, la matriu de sortida "transf", sota la condició que la imatge ha de tindre una resolució de 1280x720 píxels (OpenCV dev team, 2015).

Figura 6: Frame qualsevol després d'aplicar el canvi de perspectiva

```
[33]    return frame
```

Finalment s'utilitza la comanda **return**. Aquest comando fa que al executar-se aquesta línia, la funció general, en aquest cas "captura(cam,M)", finalitzi i retorni un valor o variable que pugui ser d'utilitat. Aquesta variable, la qual s'ha anomenat "frame", serà el *frame* capturat però amb la perspectiva ja transformada (Python Foundation, 2015).
Per tant, gràcies a la funció captura(cam,M), cada *frame* que capturi la càmera serà automàticament transformat a una perspectiva aèria i guardat sota el nom "frame" per a poder ser utilitzat dins l'algoritme general.

5.5 Funció per a seguir el moviment d'un vehicle

Ara cal definir la funció més important i complexa de l'algoritme. Aquesta funció ha de reconèixer quan un vehicle ha entrat a l'escenari mitjançant la comparació entre un *frame* i el següent, a la vegada que filtra altres diferències que poden aparèixer, com per exemple el moviment d'una persona o un animal dins el pàrquing. Després de seguir-ne el moviment ha de determinar on ha estacionat i guardar certa informació, com per exemple la situació del punt central del vehicle, la seva llargada i amplada, o l'angle de rotació de l'espai que ocupa. Per altra banda, també ha de reconèixer quan un vehicle ja estacionat, es mou per a sortir del pàrquing, eliminant la posició d'aquell vehicle de l'escenari.

La funció completa, en aquest cas anomenada "segueixcotxe" és:

```
[34] def segueixcotxe(frame):
[35]     bg=cv2.BackgroundSubtractorMOG()
[36]     bg.apply(frame)
[37]     cm0 = None
[38]     while True:
[39]             time.sleep(5)
[40]             frame = captura(cam,M)
[41]             df2=bg.apply(frame)
[42]             kernel=np.ones((12,12),np.uint8)
[43]             df2=cv2.morphologyEx(df2,cv2.MORPH_CLOSE,
                 kernel,iterations=2)
[44]             contours,hierarchy=cv2.findContours(df2,cv2.RETR_TREE,
                 cv2.CHAIN_APPROX_SIMPLE)
[45]             sx = 0.0
[46]             sy = 0.0
[47]             sa = 0.0
[48]             for c in contours:
[49]                     a = cv2.contourArea(c)
```

```
[50]                    if a < 80000: continue
[51]                    x,y=cv2.minAreaRect(c)[0]
[52]                    sx += x * a
[53]                    sy += y * a
[54]                    sa += a
[55]            if sa == 0: continue
[56]            cm1 = np.array((sx,sy))/sa
[57]            if cm0 != None and np.max(abs(cm1-cm0))< 300:
[58]                    cntG = [c for c in contours if cv2.contourArea(c) > 80000]
[59]                    assert len(cntG) == 1
[60]                    xy,wh,angle = cv2.minAreaRect(cntG[0])
[61]                    return (frame, np.array(xy), wh, angle)
[62]            cm0 = cm1
```

Per a executar aquesta funció és necessari únicament entrar-li la variable "frame", la qual és el *frame* ja transformat de la superfície útil del pàrquing.

```
[35]    bg=cv2.BackgroundSubtractorMOG()
```

Dins de la funció, la primera línia de codi és la 35, on es troba la funció **cv2.BackgroundSubtractorMOG()**. Aquesta funció, provinent de la llibreria de visió artificial OpenCV és un potent mètode per a analitzar i processar possibles canvis que poden aparèixer en un escenari. Per a fer-ho, basa el seu funcionament en un algoritme de segmentació Gaussià mesclat entre l'anàlisi del fons i del primer pla de la imatge. Fa servir un mètode per modelar cada píxel del fons de la imatge mitjançant una mescla K de distribucions gaussianes. Aquesta K, que pot anar de 3 a 5, representa els pesos de la mescla, sent cada pes la proporció de temps que cada color està a l'escena. Els colors del possible fons de la imatge seran els que estan més temps i més estàtics (Mordvintsev & Abid, 2014).

Per a iniciar la funció, es crea un objecte de tipus **cv2.BackgroundSubtractorMOG()**, en aquest cas anomenat "bg", i se li passa el "frame".

```
[36]    bg.apply(frame)
```

A la línia 36 es fa servir la comanda de Python **.apply()**. Aquest es fa servir per aplicar una certa funció (l'objecte de la qual anirà indicat abans del punt) a una variable, en aquest cas el *frame* que entra a la funció "segueixcotxe()" (Python Foundation, 2015). Per tant, en aquesta línia s'aplicarà la funció

cv2.BackgroundSubtractorMOG() al "frame". En aplicar això per primera vegada, l'objecte "bg" guardarà com a fons de la imatge pràcticament la totalitat d'aquell *frame* que transforma. Això serà de gran utilitat, ja que després, al capturar un altre *frame* dins la funció "segueixcotxe()" i aplicar "bg", aquest nou *frame* serà comparat amb l'anterior, el qual serveix com a hipotètic fons de la imatge.

Figura 7: Frame agafat com a background

Figura 8: Frame posterior

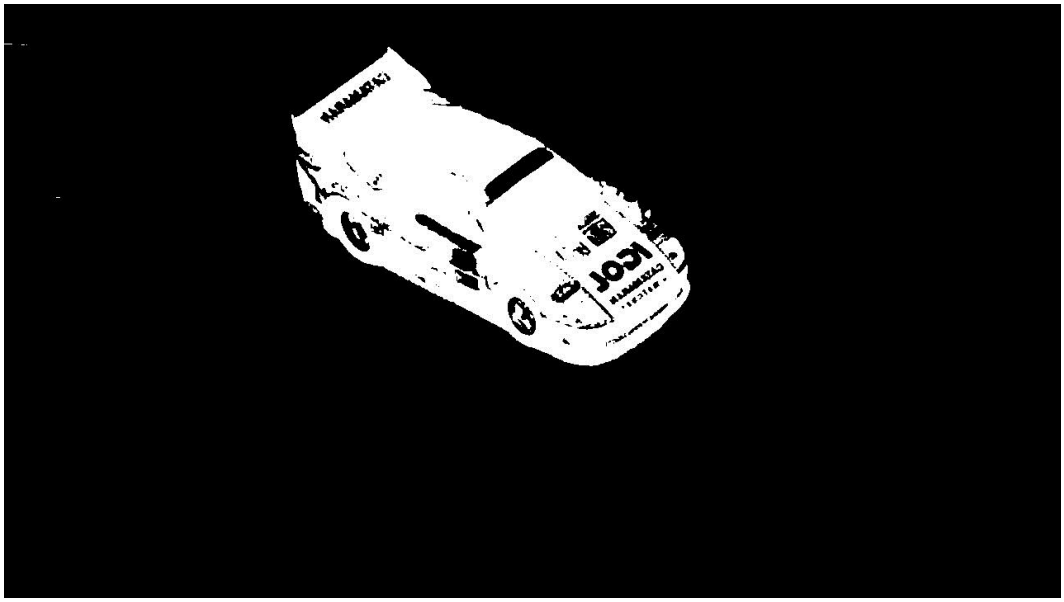

Figura 9: Exemple de diferència del nou frame respecte el background

[37] cm0 = None

En aquesta línia es fixa el valor de "cm0" com a inexistent, ja que aquest valor serà definit més endavant. Cal fixar-lo anteriorment, ja que sinó el programa, el primer cop que utilitza aquest valor, al no haver estat encara introduït, no es podria executar. "cm0" fa referència al centre de masses de l'àrea de la superfície que ha canviat en l'escenari respecte el *frame* anterior. Aquest terme s'explica amb més profunditat més endavant, el qual serveix per a comparar la coordenada d'aquest centre entre un *frame* i el seguent podent així determinar quan ha estacionat.

A la línia 38, comença el llaç principal de la funció "segueixcotxe()", el qual determinarà quan un vehicle ha entrat o sortit de l'escenari. Aquest llaç finalitza a la línia 62.
Dins el llaç, el primer que es troba és un **time.sleep()** de cinc segons. Això permet que el programa s'esperi cinc segons abans d'executar la següent línia de codi. En aquest cas la següent línia és la 40, on es crida la funció "captura()", la qual retorna un nou "frame" ja amb la perspectiva transformada. Els cinc segons d'espera anteriors fan que el programa realitzi una captura de l'escenari cada cinc segons, permetent reduir la quantitat de "frames" a processar. Aquesta seqüència de captura és suficient per a determinar diferències singulars a l'escenari.

[41] df2=bg.apply(frame)

Una vegada el programa ja té el valor del nou *frame*, se li aplica el **cv2.BackgroundSubtractorMOG()** i s'iguala a la variable anomenada "df2". La imatge resultant és de tipus binari (blanc i negre), on els canvis respecte el *frame* anterior apareixen en blanc, i el fons en negre.

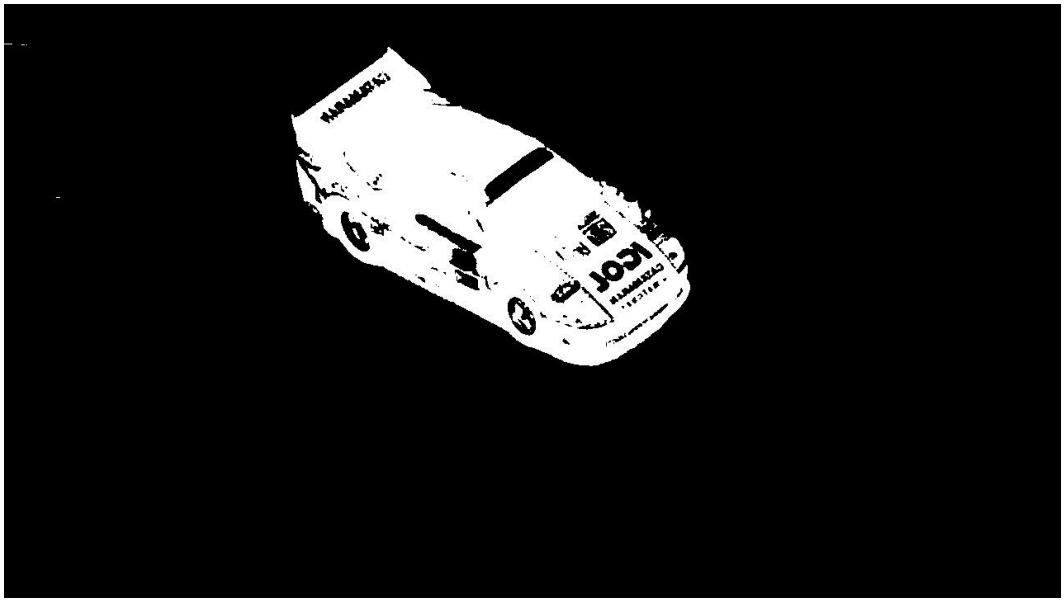

Figura 10: Resultat d'aplicar cv2.BackgroundSubstractorMOG()

```
[42]    kernel=np.ones((12,12),np.uint8)
[43]    df2=cv2.morphologyEx(df2,cv2.MORPH_CLOSE,
        kernel,iterations=2)
```

Les línies 42 i 43, serveixen per a compactar les diferències que es mostren a l'escenari, eliminar el "soroll" d'una imatge. Aquest "soroll" són petites diferències, degudes per exemple a canvis en la lluminositat. Aquest soroll es fa per exemple, que un vehicle no es mostri com un objecte compacta i en color blanc després d'aplicar-li el **cv2.BackgroundSubtractorMOG().** Això dificulta l'anàlisi dels contorns existents a la imatge. És per això que a la línia 43 es fa servir la funció **cv2.morphologyEx()**. Aquesta funció aplica transformacions morfològiques sobre la imatge. Donat un tipus de transformació, en aquest cas interessa un cv2.MORPH_CLOSE s'aconsegueix tancar petits forats (punts negres) que es troben dins de l'objecte en primer pla (mostrat en color blanc). El "kernel" definit a la línia anterior és necessari per a fer que només es tanquin tots els punts en negre de la superfície en blanc fins a una mida de 12.000 píxels.
A les següents imatges es mostra la diferència d'una imatge binària abans i després d'aplicar-li una operació de "closing".

Figura 11: Resultat d'aplicar una acció de "closing"

```
[44]    contours,hierarchy=cv2.findContours(df2,cv2.RETR_TREE,
        cv2.CHAIN_APPROX_SIMPLE)
```

Seguidament, a la línia 44 s'utilitza la funció **cv2.findContours()**. Cada contorn és un conjunt continu de punts que tenen el mateix color. Aquesta funció treballa molt millor sobre una imatge binaria, ja que aquesta anàlisi de punts és més fàcil. Per a utilitzar la funció, cal introduir tres arguments. El primer és la imatge que es vol analitzar, "df2", el segon és amb quin mode es vol recuperar o organitzar els contorns, en aquest cas s'ha utilitzat el mode **cv2.RETR_TREE** , el qual recupera tots els contorns i els reconstrueix seguint una jerarquia completa. El tercer argument és quin mètode es vol utilitzar per a aproximar els contorns. S'ha utilitzat el mètode **cv2.CHAIN_APPROX_SIMPLE** , el qual comprimeix els segments horitzontals, verticals i diagonals, i en registra només els punts finals.
Aquesta funció retorna els contorns detectats, guardats cada un com un vector de punts, anomenats en aquest cas "contours", i un vector que conté informació sobre la tipologia de la imatge, en aquest cas anomenat "hierarchy", el qual no s'utilitzarà en aquest programa (OpenCV dev team, 2015).
Cada contorn serà cada canvi que hi ha hagut sobre l'escenari respecte al *frame* anterior.

```
[45]    sx = 0.0
[46]    sy = 0.0
[47]    sa = 0.0
```

Seguidament, a les línies 45, 46 i 47, s'inicialitzen a zero els sumatoris de coordenades x, y i àrees. Utilitzats a continuació.

```
[48]        for c in contours:
[49]            a = cv2.contourArea(c)
[50]            if a < 80000: continue
[51]            x,y=cv2.minAreaRect(c)[0]
[52]            sx += x * a
[53]            sy += y * a
[54]            sa += a
```

A la línia 48 es crea un **for** que recorre cada contorn guardat sota la variable "contours". De cada contorn, es guarda la seva àrea, utilitzant la funció **cv2.contourArea()**. Aquesta funció calcula l'àrea del contorn introduït, retornant un valor corresponent al nombre de píxels de l'àrea (OpenCV dev team, 2015).

```
[50]            if a < 80000: continue
```

A la línia 50 s'utilitza un **if** per a filtrar les àrees de menor mida, que podrien correspondre a variacions en la lluminositat a l'escenari, o moviments de persones o animals. S'ha fixat que aquests canvis, tenen una àrea aproximada sobre l'escenari de menys de 80000 píxels, per tant, si el programa analitza un contorn amb àrea més petita que 80000 píxels, gràcies al comando **continue**, no executarà la següent línia, sinó que tornarà a la línia 48, analitzant el següent contorn.
Cal remarcar que la condició de 80000 píxels varia depenent de la posició i altura on es col·loqui la càmera, així com de la il·luminació de l'escenari. Aquest valor s'hauria de calibrar cada vegada que es volgués instaurar el sistema de visió artificial en un pàrquing real per a un correcte funcionament d'aquest.

Quan un dels contorns tingui àrea més gran que 80000 píxels, el programa executarà la línia 51.

```
[51]            x,y=cv2.minAreaRect(c)[0]
```

En aquesta línia s'utilitza la funció **cv2.minAreaRect()**. Aquesta retorna una estructura de caixa en dues dimensions on s'encabeix el contorn analitzat. Aquesta caixa es crea utilitzant el punt central del contorn, l'amplada i llargada aproximada del contorn, i l'angle de rotació (OpenCV dev team, 2015).
En aquest punt del programa, només és necessari guardar els valors x i y referents a la coordenada del punt central del contorn. És per això que al costat

de la funció, es defineix un claudàtor amb un zero enmig. Aquest indica que únicament es vol guardar els valors del primer vector que retorna **cv2.minAreaRect()** que és el punt central del contorn.

Arribat aquest punt de l'algoritme, cal explicar el mètode que s'ha desenvolupat per a què l'algoritme analitzi els dos possibles canvis en l'escenari que poden succeir:

- Entrada d'un nou vehicle: Al entrar un vehicle nou, només existirà un contorn de diferència a l'escenari respecte al *frame* anterior, el qual serà pres com a fons de la imatge.
- Sortida d'un vehicle estacionat: Al sortir un vehicle, apareixeran dos contorns de diferència a l'escenari respecte al *frame* anterior, un en el lloc on abans hi havia el vehicle, i l'altre on el vehicle s'ha mogut a la nova localització.

Degut a això s'ha creat un sistema per a poder guardar el centre de masses de les àrees dels contorns que han aparegut a l'escenari. Aquest centre, serà processat més endavant per a determinar quan un vehicle ha estacionat o ha sortit de l'espai del pàrquing.

Càlcul del centre de masses quan apareix un vehicle a l'escenari

A continuació es mostren dos *frames* capturats un rere l'altre. En el primer no hi ha hagut cap canvi respecte a l'anterior, però en el segon *frame* ha aparegut un vehicle a l'escenari.

Figura 12: Frame 1 - Pàrquing buit

Figura 13: Frame agafat com a background

Figura 14: Frame 2 - apareix un vehicle

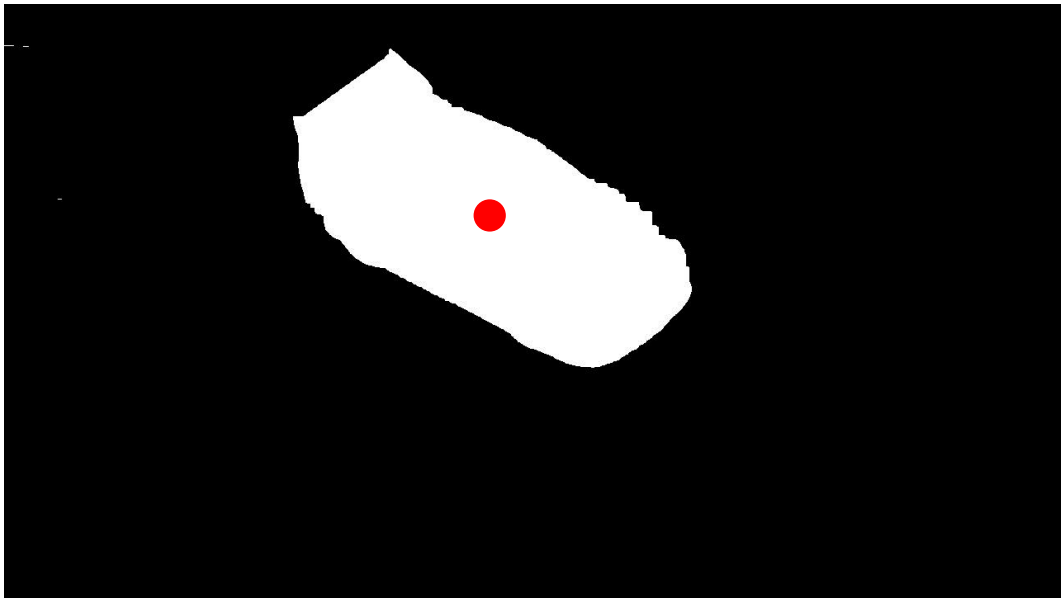

Figura 15: Diferències del frame 2 respecte el background

```
[48]        for c in contours:
[49]            a = cv2.contourArea(c)
[50]            if a < 80000: continue
[51]            x,y=cv2.minAreaRect(c)[0]
[52]            sx += x * a
[53]            sy += y * a
[54]            sa += a
[55]        if sa == 0: continue
[56]        cm1 = np.array((sx,sy))/sa
```

Si s'analitza línia per línia el que fa l'algoritme per cada *frame* llegit, en el primer no hi apareix cap contorn, per tant el sumatori de coordenades x i y (sx i sy) i el sumatori d'àrees (sa) seguirà sent 0.0 . Com que "sa" és zero, al complir la condició del **if** a la línia 55, al executar-se la comanda **continue**, el programa tornarà a entrar al **while** de la línia 38 i capturarà un nou *frame*.

En el segon *frame* ja existeix un contorn. Per tant ara el programa si executarà la línia 49 i posteriors, ja que a més a més es suposa que aquest contorn és més gran que 80000 píxels. En "sx" quedarà guardat el valor de la coordenada "x" del punt central del contorn multiplicat per l'àrea. El mateix succeeix per a la coordenada "y". El sumatori d'àrees ("sa"), en aquest cas serà simplement la pròpia àrea de l'únic contorn trobat.

En haver únicament un contorn, el llaç **for** ja haurà finalitzat. Per tant, a la línia 56 es calcularà el centre de masses d'aquest contorn, creant un vector on la primera component serà la divisió (sx/sa) i la segona (sy/sa). Aquest vector és

essencial per a poder determinar finalment la posició del vehicle estacionat. Aquest centre de masses quedarà guardat com a "cm1".
Els centre de masses s'indica a les figures amb un punt vermell.

```
[55]        if sa == 0: continue
[56]        cm1 = np.array((sx,sy))/sa
[57]        if cm0 != None and np.max(abs(cm1-cm0))< 300:
[58]            cntG = [c for c in contours if cv2.contourArea(c) > 80000]
[59]            assert len(cntG) == 1
[60]            xy,wh,angle = cv2.minAreaRect(cntG[0])
[61]            return (frame, np.array(xy), wh, angle)
[62]        cm0 = cm1
```

Seguidament, a la línia 57, el programa no passaria a executar la línia 58, ja que no compleix la condició "np.max(abs(cm1-cm0))< 300". La condició "cm0 != None" si que la compliria, ja que s'ha d'entendre que no és el primer cop que el programa s'executa, per tant "cm0" portaria un valor emmagatzemat.
La segona condició no es compliria, ja que amb tota seguretat, la màxima diferència en valor absolut entre el "cm0" (centre de masses del contorn del *frame* anterior) i el "cm1" de l'actual seria més gran que 300.
Per tant, el programa passarà directament a executar la línia 62, refrescant el valor de "cm0" per el nou valor de "cm1".
A continuació es captura un tercer *frame*:

Figura 16: Frame 3 - el vehicle ha estacionat

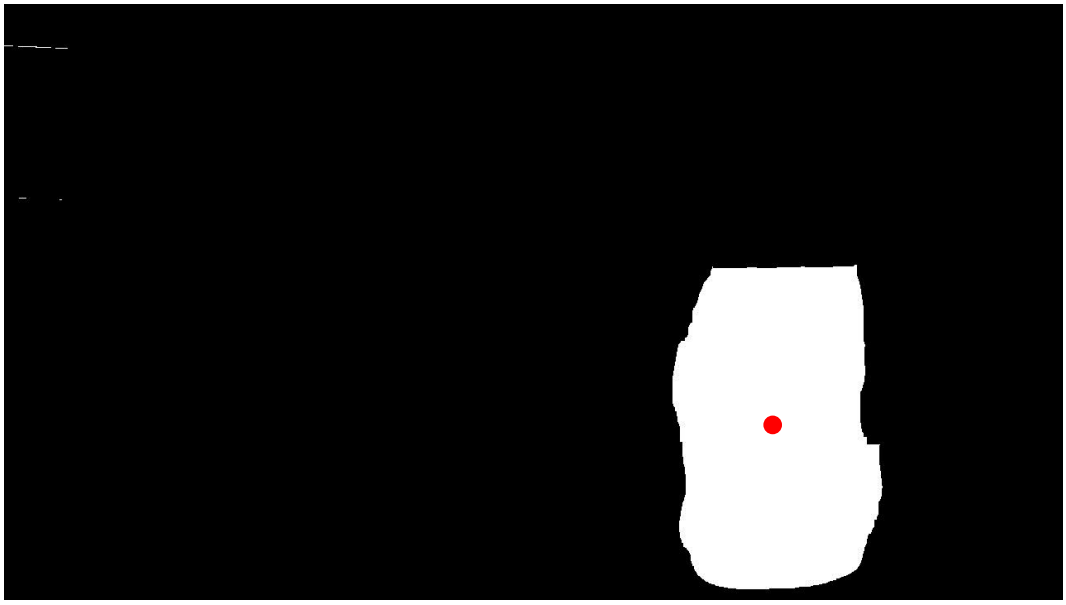

Figura 17: Diferències del frame 3 respecte el background

En aquest, el vehicle ha estacionat. Els valors de "sx", "sy" i "sa" serien únicament els d'aquest nou contorn , ja que anteriorment, al acabar d'executar la línia 62 s'ha tornat a la línia 38; s'ha capturat un nou *frame* i s'han tornat a inicialitzar a 0.0 els valors "sx","sy" i "sa". Una vegada obtinguts els nous valors dels sumatoris, s'ha calculat el nou "cm1".
Ara, a la línia 57, el programa no compleix tampoc les condicions, ja que la diferència de "cm1" i el "cm0" anterior és molt gran.

No obstant, si es captura un quart *frame* , el qual seria igual a l'anterior ja que el vehicle ja ha estacionat i no s'ha mogut, ara si es compleix les condicions del **if**, ja que el nou "cm1" i el "cm0" anterior son molt similars o iguals.

Figura 18: Frame 4 - el vehicle segueix estacionat

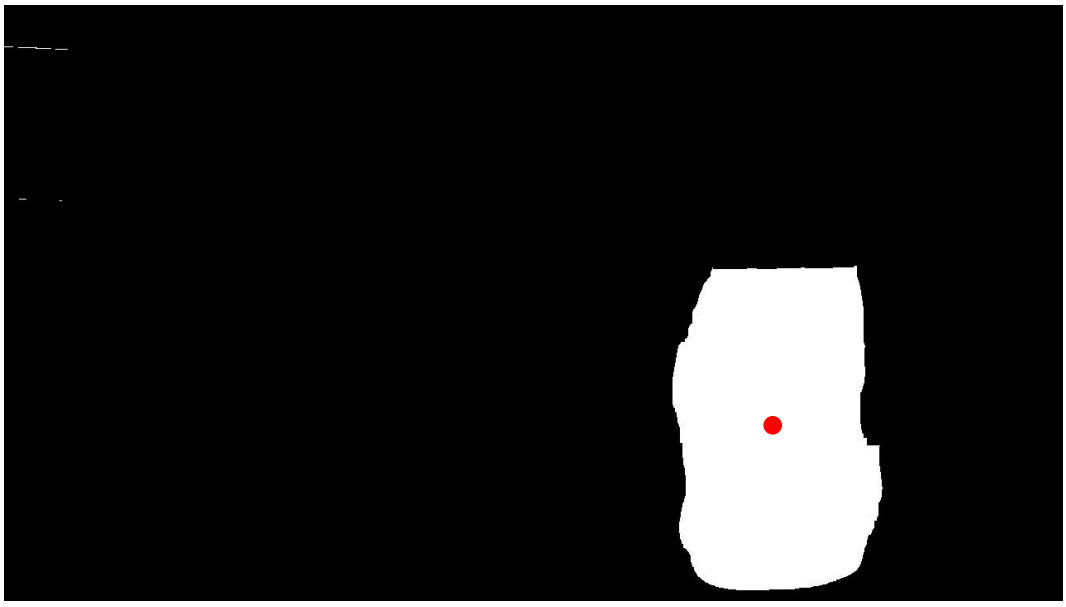

Figura 19: Diferències del frame 4 respecte el background

```
[58]    cntG = [c for c in contours if cv2.contourArea(c) > 80000]
[59]    assert len(cntG) == 1
```

A la línia 58, es crea un comptador utilitzant un **for** que recorre tots els contorns existents. A la línia 59, s'aplica un comando **assert** per assegurar que només hi ha un contorn a l'escenari (de més de 80000 píxels d'àrea) quan el vehicle ha estacionat. Aquesta condició és necessària per fer funcionar l'algoritme en el cas que un vehicle surti, com s'explica més endavant. També és necessari per a fer

funcionar el programa sota la condició d'analitzar un sol vehicle que entra o surt, per tant donaria errors si existís més d'un vehicle movent-se a l'escenari.

```
[60]        xy,wh,angle = cv2.minAreaRect(cntG[0])
[61]        return (frame, np.array(xy), wh, angle)
```

Finalment, a la línia 60, es torna a utilitzar la funció **cv2.minAreaRect()** simplement per emmagatzemar la informació exacta del punt central, amplada i llargada, i angle de rotació de la capsa virtual que encabeix el vehicle estacionat.
A l'última línia, la 61, es fa servir la comanda **return** per retornar la informació esmentada anteriorment, la qual determinarà la posició exacta sobre la superfície del pàrquing del nou vehicle que ha estacionat. Cal retornar la primera tupla en forma de vector per a poder comparar aquest valor dins la llista de cotxes registrats, explicada més endavant.
A part, es retorna també l'últim frame captura, el qual s'usa en el llaç principal per a mostrar-lo per pantalla tot indicant la segmentació de cada vehicle a l'espai.

Càlcul del centre de masses quan un vehicle està sortint de l'escenari

El cas d'un vehicle estacionat anteriorment però movent-se per a sortir de l'escenari és més complexa d'analitzar. És per això que s'ha adequat el sistema de determinació del centre de masses dels contorns de l'escenari per a poder determinar quin vehicle ha abandonat el pàrquing.

Per a explicar el sistema es parteix d'un *frame* on hi ha un vehicle estacionat. Aquest no apareix en blanc sobre el fons de la imatge, ja que en el *frame* anterior aquell vehicle seguia estacionat en el mateix lloc, per tant no ha existit cap canvi a l'escenari. Tot i això, per a explicar el funcionament del sistema, es marca on està estacionat el vehicle utilitzant un requadre vermell.

Figura 20: Frame 1 - Pàrquing amb un vehicle estacionat

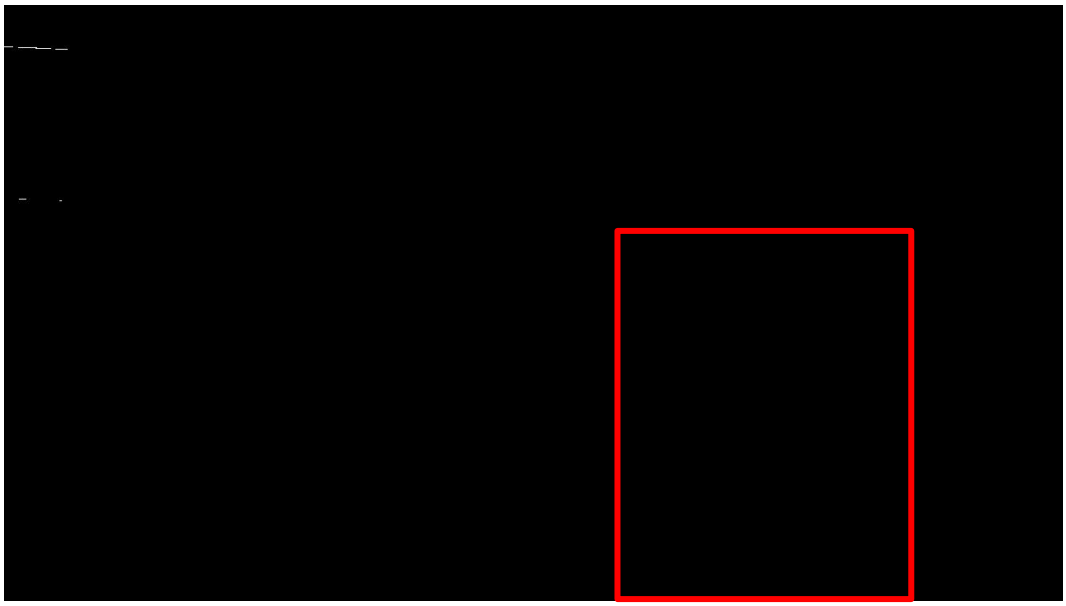

Figura 21:Frame agafat com a background amb la posició marcada del vehicle estacionat

Per tant, en analitzar el següent *frame*, si aquest vehicle que abans estava estacionat s'ha mogut de lloc, ara no apareix una sola diferència i per tant un sol contorn, sinó que n'apareixeran dues. Una en el lloc on abans hi havia el vehicle i l'altre en la posició on es trobi actualment.

Figura 22: Frame 2 - el vehicle es mou per abandonar el pàrquing

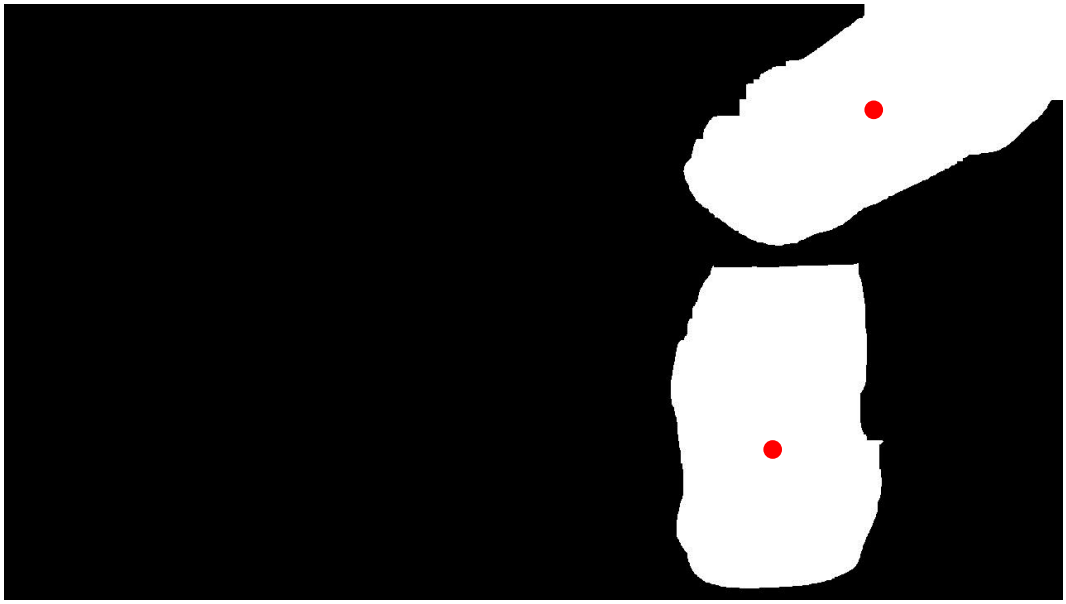

Figura 23: Diferències del frame 2 respecte el background

Aquí és on el sistema de centre de masses definit anteriorment és realment útil.

```
[48]        for c in contours:
[49]            a = cv2.contourArea(c)
[50]            if a < 80000: continue
[51]            x,y=cv2.minAreaRect(c)[0]
[52]            sx += x * a
[53]            sy += y * a
[54]            sa += a
```

```
[55]        if sa == 0: continue
[56]        cm1 = np.array((sx,sy))/sa
```

Ara existeix més d'un contorn a l'escenari. En recórrer el primer contorn quedarà guardat un valor de la coordenada "x" per l'àrea del contorn en la variable "sx", un de la coordenada "y" per l'àrea en "sy" i la seva àrea en "sa". Seguidament el programa torna a la línia 48 per a recórrer el segon contorn. Ara, la variable "sx" serà el sumatori de la "sx" anterior més la coordenada "x" del punt central per l'àrea del segon contorn. El mateix passa amb la nova variable "sy". La nova variable "sa" serà el sumatori de les dues àrees.
Finalment, a la línia 56, es crearà un "cm1" que realment serà el centre de masses del conjunt format pels dos contorns registrats.
Cal remarcar que es multiplica l'àrea del contorn per la coordenada com a mesura de rectificació, ja que pot ser que degut a la diferent posició i rotació entre un contorn i l'altre, un tingui una àrea significativament més gran, cosa que podria afectar el càlcul del punt de centre de masses del conjunt.

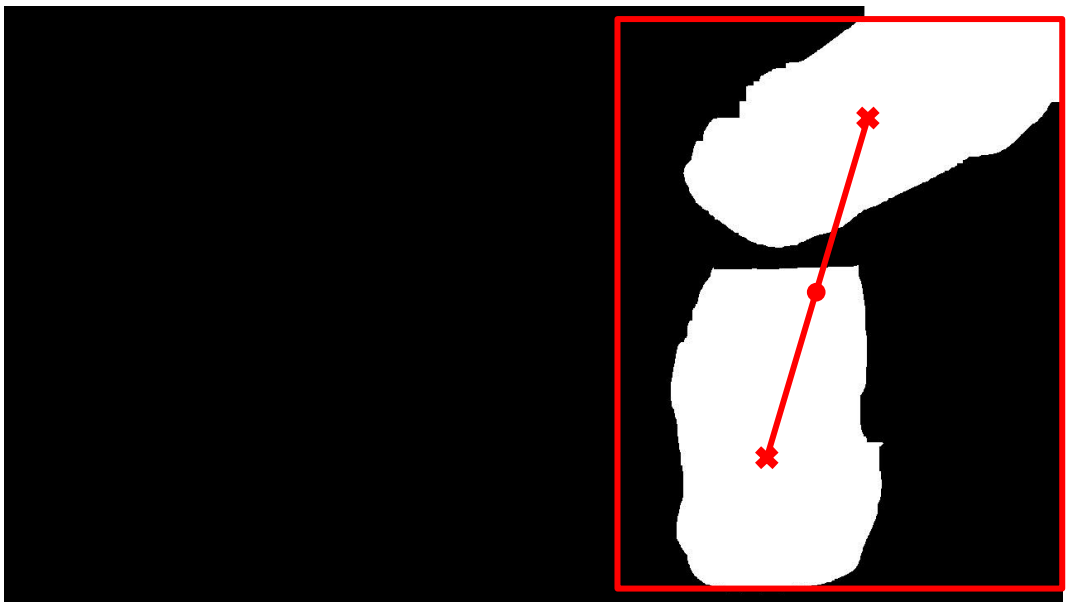

Figura 24: Centre de masses del conjunt de diferències respecte el background

El centre de masses d'aquest conjunt de doc contorns es mostra a la figura com el punt vermell situat a la recta que uneix els dos centres dels contorns existents a l'escenari.
Aquest "cm1" registrat queda guardat sota la variable "cm0" a la línia 62.
En la següent figura, es mostra un frame posterior, on el vehicle segueix movent-se per arribar a la sortida.

Figura 25: Frame 3 - el vehicle segueix movent-se

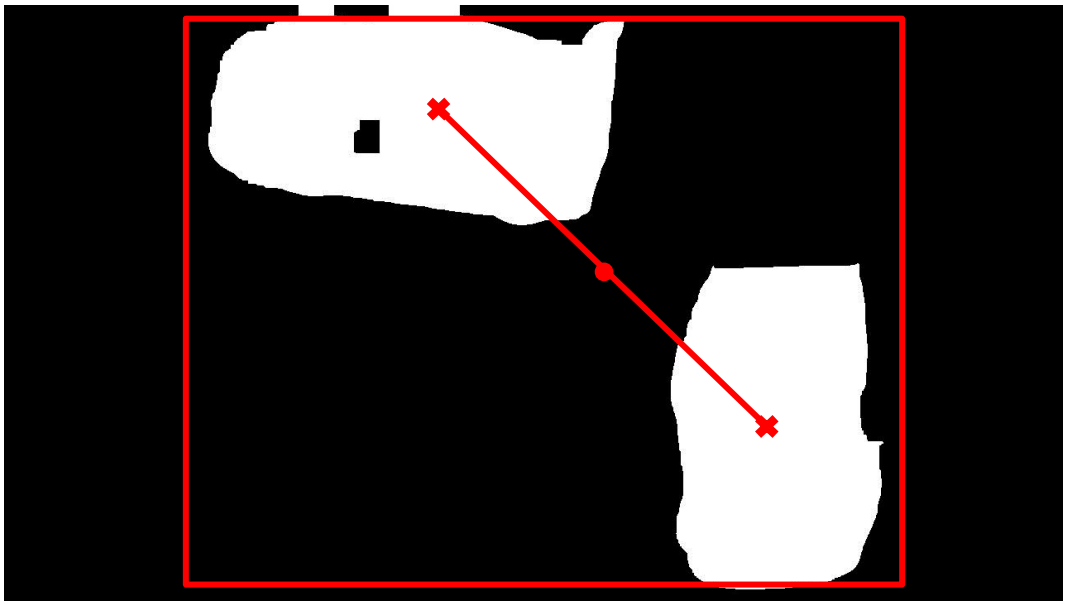

Figura 26: Centre de masses del conjunt de diferències respecte el background

En aquest *frame* es pot veure com el vehicle s'està allunyant encara més de la seva posició original d'estacionament. Per tant, ara s'haurà creat un nou centre de masses "cm1".

```
[57]        if cm0 != None and np.max(abs(cm1-cm0))< 300:
```

A la línia 57, es compara el nou "cm1" amb el centre de masses del conjunt del *frame* anterior ("cm0"). Al ser la diferència major que deu unitats, no complirà la

condició imposada per el **if** i el programa executarà la línia 62, guardant el nou "cm1" com a "cm0".

A continuació es mostra un següent *frame* on el vehicle ja ha abandonat el pàrquing, per tant a l'escena, comparant aquest *frame* amb l'inicial (agafat com a fons o *background*) només apareixerà el contorn de la zona buida d'estacionament que ha deixat.

Figura 27: Frame 4 - el vehicle ha abandonat el pàrquing

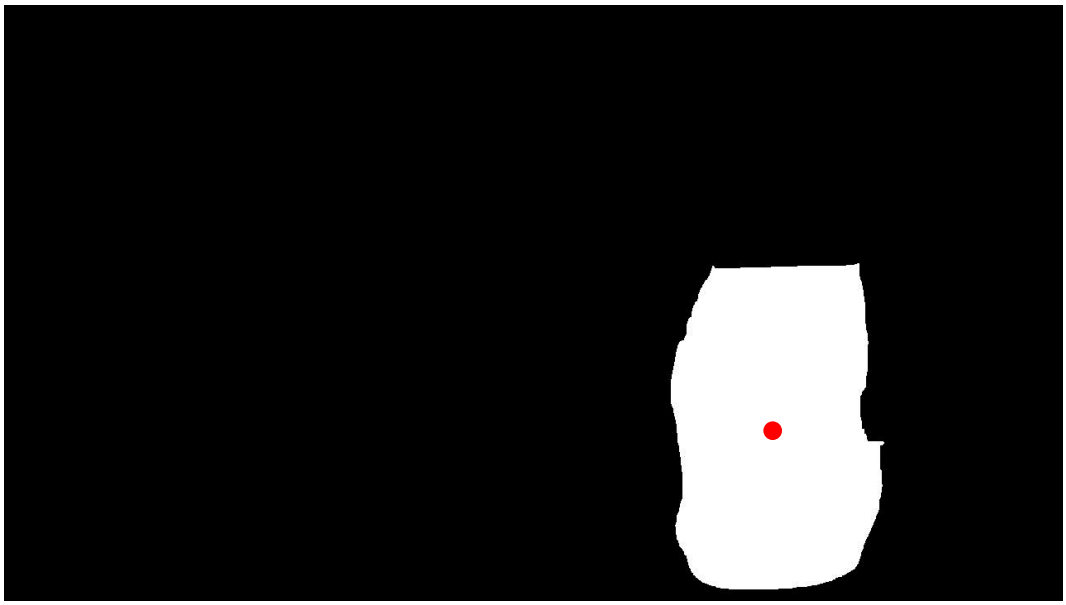

Figura 28: Diferències del frame 4 respecte el background

Ara tant sol hi ha un contorn a l'escenari. Tot i això quan el programa executa la línia 57, tampoc es compleix la condició imposada per el **if**, ja que la diferència entre el "cm1" actual i el "cm0" anterior será major que deu.
Seguidament, es guarda aquest "cm1" com a "cm0" i es torna a la línia 39.
Al capturar un nou *frame* cinc segons després de l'anterior, l'escenari es manté igual que l'anterior. Per tant ara, en trobar el nou "cm1", aquest és igual o pràcticament igual que l'anterior (el qual és "cm0").

Figura 29: Frame 5 - el pàrquing segueix buit

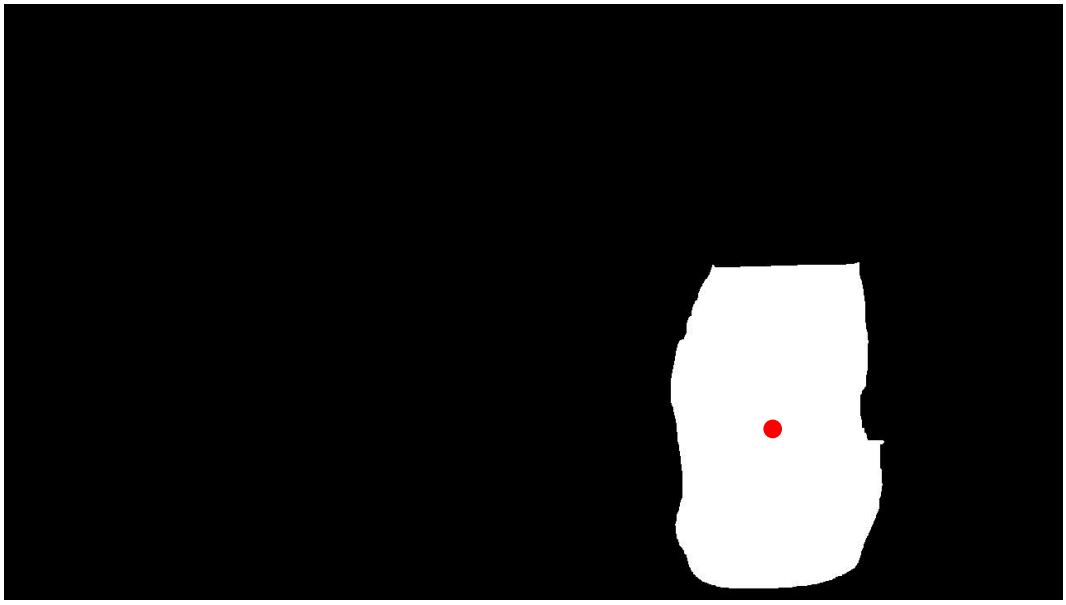

Figura 30: Diferències del frame 5 respecte el background

En executar-se la línia 57, la condició de "np.max(abs(cm1-cm0))< 300" ja es compleix i el programa executarà la línia 58 i posteriors.
Per tant, ara el programa retorna el punt central, amplada i llargada, i angle de rotació del contorn existent a l'escenari, en aquest cas l'espai buit que ha deixat el vehicle que ha marxat.
Aquesta informació es fa servir en el llaç principal per a eliminar un vehicle que havia estat registrat dins una llista on apareixen les coordenades de cada vehicle que ha estacionat, explicat més endavant.

5.6 Creació de la llista de vehicles registrats

La finalitat de l'algoritme és poder determinar la localització exacta i espai que ocupa cada vehicle sobre la superfície del pàrquing.

```
[63] registreCotxes=[]
```

Per tant, s'ha creat una llista buida anomenada "resgistreCotxes" on es guardarà, en cada posició, la coordenada (x,y) del punt central del vehicle estacionat, l'amplada i llargada del requadre que l'encabeix, i l'angle de rotació d'aquest requadre. Més endavant, aquesta informació s'utilitza per poder segmentar virtualment l'espai que ocupa cada vehicle estacionat dins el pàrquing.

5.7 Llaç principal

El llaç principal determina l'ordre global que segueix el programa en executar-se. És on es criden totes les funciones definides anteriorment per a què tot funcioni correctament i finalment s'aconsegueixi mostrar la superfície del pàrquing segmentat amb cada vehicle estacionat.

```
[64] k=0
[65] while 1:
[66]     frame = captura(cam,M)
[67]     parquing, xy, wh, ang = segueixcotxe(frame)
[68]     treure = False
[69]     for a in registreCotxes:
[70]             if np.max(abs(a[0]-xy)) < 300:
[71]                     registreCotxes.remove(a)
[72]                     treure = True
[73]                     break
[74]     if not treure:
```

```
[75]            registreCotxes.append((xy,wh,ang))
[76]    for xy,wh,ang in registreCotxes:
[77]            rect = tuple(xy), wh, ang
[78]            box=cv2.cv.BoxPoints(rect)
[79]            box=np.int0(box)
[80]            cv2.drawContours(parquing,[box],-1,(0,0,255),2)
[81]    k+=1
[82]    cv2.imwrite('Desktop/escena%02d.jpg'%k,parquing)
```

Abans d'entrar al llaç principal cal crear una variable "k", inicialitzada amb el valor zero. Aquesta variable servirà per enumerar cada imatge de l'escenari que es guardarà cada vegada que un vehicle estacioni o surti del pàrquing.

El llaç comença a la línia 65 amb un **while** que s'executa infinitament.

```
[66]    frame = captura(cam,M)
```

A la línia 66 es captura i es transforma la perspectiva del primer *frame*. Aquest "frame" es fa servir dins la funció "segueixcotxe()" com a fons o *background* per a comparar qualsevol *frame* capturat dins la funció "segueixcotxe()" amb aquest. Cada vegada que el programa surti de la funció "segueixcotxe()" i es completi el llaç, es torna a agafar un següent *frame* que servirà com a *background*.

```
[67]    parquing, xy, wh, ang = segueixcotxe(frame)
```

A la línia 67 es crida la funció "segueixcotxe(frame)". Quan un vehicle hagi estacionat o marxat de l'escenari, la funció retornarà en primer lloc l'últim *frame* capturat re nombrat com a "parquing", la coordenada del punt central del vehicle o l'espai buit que un ha deixat com a "xy", l'amplada i llargada del requadre que l'encabeix com a "wh" i l'angle d'aquest com a "ang".

```
[68]    treure = False
[69]    for a in registreCotxes:
[70]            if np.max(abs(a[0]-xy)) < 300:
[71]                    registreCotxes.remove(a)
[72]                    treure = True
[73]                    break
```

A la línia 68 s'inicialitza la variable booleana anomenada "treure" com a falsa. Aquesta booleana serveix per a indicar si el vehicle estacionat s'ha d'incloure a

la llista "registreCotxes" o l'espai buit trobat s'utilitzarà per a eliminar les coordenades d'un vehicle ja existent a la llista.
A la línia 69 es crea un **for** per a recórrer cada posició de la llista "registreCotxes". Seguidament, a la línia 70, es crea un **if** per a imposar una condició. Aquesta condició compara el primer vector (coordenada del punt central) de la posició de la llista que s'estigui recorrent en aquell moment amb la coordenada del punt central que ha retornat la funció "segueixcotxe()" a la línia 67 ("xy"). Si la diferència és més petita que 300, significarà que la diferència entre un punt i l'altre és pràcticament inexistent i per tant es tracta de la situació on un vehicle ja registrat anteriorment ha abandonat l'escenari.

```
[71]                registreCotxes.remove(a)
[72]                treure = True
```

Una vegada determinat que la posició retornada per la funció "segueixcitxe()" és la de l'espai buit que un vehicle ha deixat en abandonar el pàrquing, a la línia 71 s'elimina aquella posició de la llista de vehicles registrats utilitzant la comanda **remove**.
Seguidament, es canvia el valor del booleà "treure" a cert. Això és necessari per a no confondre el programa. A la línia 73, s'executa un **break** per a sortir del llaç **for**, permetent que el programa es segueixi executant sense haver de seguir comparant tos els vehicles registrats. Cosa que millora el rendiment del programa.

```
[74]    if not treure:
[75]        registreCotxes.append((xy,wh,ang))
```

Com que el booleà ha canviat a cert, no passa la condició de la línia 74.
Contràriament, si la posició retornada per la funció "segueixcotxe()" hagués retornat les coordenades d'un vehicle nou estacionat a l'escenari, la condició de la línia 70 no es compleix, ja que no hi haurà cap vehicle registrat amb una posició pràcticament igual. Per tant el booleà "treure" es mantindria fals i la condició de la línia 74 si es compliria. En aquest moment, el programa executaria la línia 75 afegint tota la informació del nou vehicle estacionat a l'última posició de la llista "registreCotxes" gràcies a la comanda **append**.

```
[76]    for xy,wh,ang in registreCotxes:
[77]        rect = tuple(xy), wh, ang
[78]        box=cv2.cv.BoxPoints(rect)
[79]        box=np.int0(box)
[80]        cv2.drawContours(parquing,[box],-1,(0,0,255),2)
```

Les línies 76 a 80 serveixen per a dibuixar virtualment el requadre que encabeix cada vehicle a l'escenari.
Es comença amb un **for** que recorre cada posició de la llista "registreCotxes" utilitzant tota la informació emmagatzemada (punt central "xy", amplada i llargada "wh" i angles de rotació "ang").
Aquesta informació, s'iguala sota el nom de la variable "rect" tot canviant la forma de la coordenada "xy" a una tupla, ja que anteriorment s'ha canviat a forma vectorial per a poder fer les comparacions adients.

```
[78]        box=cv2.cv.BoxPoints(rect)
[79]        box=np.int0(box)
[80]        cv2.drawContours(parquing,[box],-1,(0,0,255),2)
```

La funció **cv2.cv.BoxPoints()** de la línia 78 serveix per a obtenir, gràcies a la informació emmagatzemada en "rect", els quatre vèrtexs del requadre que es dibuixarà. A la línia 79 es fa un canvi a nombres enters per a què la funció de la següent línia, **cv2.drawContours()** funcioni correctament.
Aquesta funció, donat una imatge d'entrada, en aquest cas l'últim *frame* capturat, i una llista amb els vèrtexs del requadre a dibuixar, dibuixa aquest requadre sobre la imatge donada. Les propietats del requadre (gruix de la línia i color) es poden modificar gràcies als números inclosos dins el parèntesi de la funció (Mordvintsev & Abid, 2014).

Gràcies al **for** de la línia 76, el programa dibuixaria tots els requadres dels vehicles existents a l'escenari sobre el mateix *frame* fins que els hagués recorregut tots.

```
[81]    k+=1
[82]    cv2.imwrite('Desktop/escena%02d.jpg'%k,parquing)
```

A la línia 81, es suma +1 al valor inicial de "k" cada vegada que s'executa aquesta línia. Això permet definir el nom de la imatge de l'escena que es guardarà cada vegada que un vehicle estacioni o abandoni el pàrquing. La primera imatge s'anomenarà "escena01", la segona "escena02" i així successivament.
Finalment, a la línia 82, s'utilitza la comanda **cv2.imwrite()** per a guardar la imatge de l'últim *frame* capturat amb tots els requadres de cada vehicle estacionat dibuixats sobre ell. Això permet una ràpida visió per part de l'usuari de la posició de cada vehicle.

Figura 31: Pàrquing segmentat

Al trobar-se tot dins un llaç principal que es va repetint indefinidament, cada vegada que es determini que un vehicle ha estacionat o ha abandonat l'escenari, es guardarà una nova imatge a la localització desitjada per l'usuari, en aquest cas l'escriptori de l'ordinador (*Desktop*).

5.8 Altres exemples del funcionament de l'algoritme

Per a mostrar l'eficiència de la funció **cv2.BackgroundSubtractorMOG()** sota qualsevol escenari per a determinar canvis sobre aquest, s'han realitzat unes proves en un escenari fictici on el fons no té un sol color, sinó una mescla de fortes tonalitats.

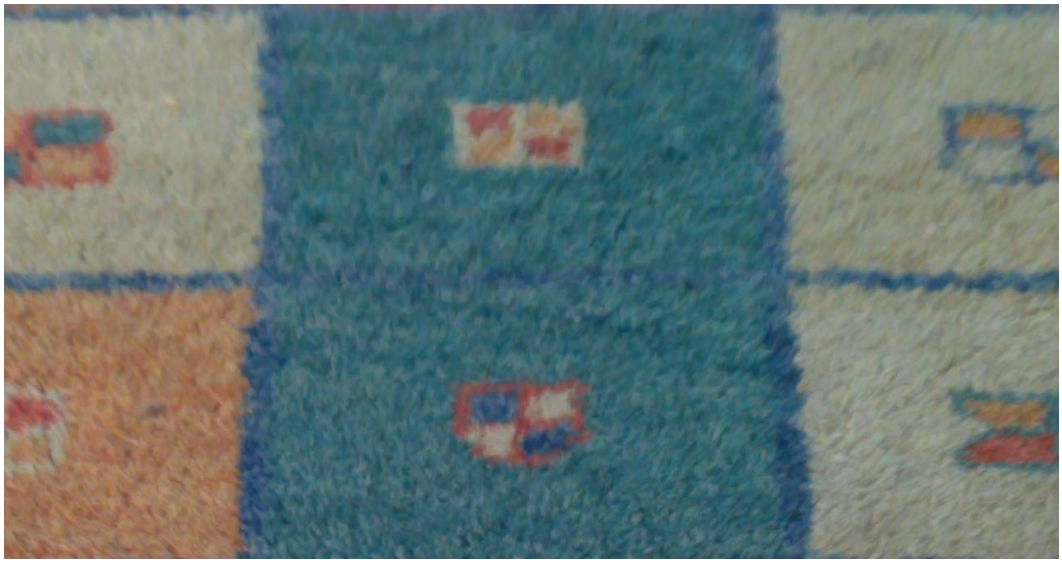

Figura 32: Escenari buit

L'escenari escollit és un tapís on es mesclen diverses tonalitats de verds, blaus, taronges i grocs.
Cal remarcar que les imatges ja es mostren amb la perspectiva transformada a zenital.

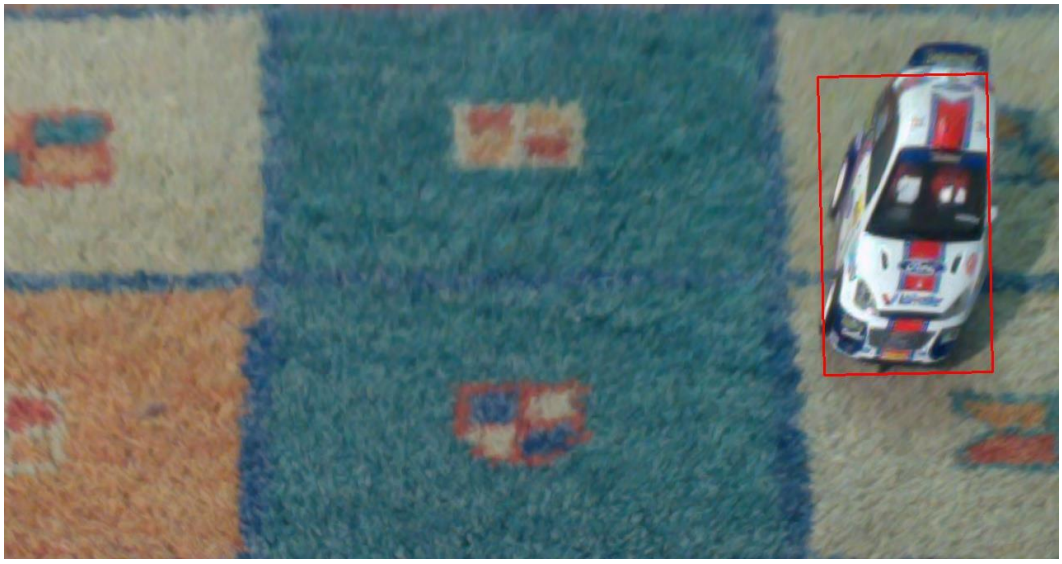

Figura 33: Escenari amb el primer vehicle estacionat marcat

Una vegada ha estacionat el primer vehicle, s'observa com l'algoritme ha determinat adequadament la superfície d'aquest.

Figura 34: Escenari amb tres vehicles estacionats marcats

En aquesta imatge es pot observar com l'algoritme determina perfectament la posició dels vehicles sobre l'escenari, independentment de la quantitat d'aquests.

Figura 35: Situació de l'escenari quan un vehicle l'ha abandonat

Aquesta imatge mostra com l'algoritme determina sense error quan un vehicle ha abandonat l'escenari i n'elimina la seva posició.

Figura 36: Escenari final

Finalment, en aquesta imatge es pot observar com l'algoritme ha eliminat correctament el segon vehicle que ha abandonat l'escenari, deixant únicament la situació de l'únic vehicle existent.

6 Aspectes econòmics

Ja que en aquest estudi es desenvolupa un prototip de la primera fase d'un programa per a gestionar l'accessibilitat en pàrquings exteriors s'escau únicament calcular un pressupost aproximat del temps invertit i els recursos materials utilitzats per a desenvolupar aquest prototip. La inversió per a realitzar aquest estudi ha estat de 9100€ sense impostos. Aquest pressupost està desglossat al document adjunt anomenat *Budget*.

El pressupost real del programa es redactaria quan aquest algoritme estigués del tot desenvolupat, incloent-hi la segona fase que és la simulació del moviment dels vehicles estacionats en temps real, sota una interfase gràfica en forma de programa informàtic.

En el punt de disposar del programa informàtic real, se'n podria estimar un preu en relació amb les hores i recursos invertits per a desenvolupar-lo. A més a més, del cost de l'ordinador Raspberry Pi i la Pi Càmera caldria afegir el cost d'una armadura hermètica on col·locar aquestes dues peces de hardware i les seves connexions, i per últim el temps necessari per a situar la càmera en un lloc elevat sobre el pàrquing i connectar tot el sistema a la xarxa elèctrica i informàtica.
La connexió a la xarxa informàtica seria necessària per a poder monitorar el programa, i per tant controlar el pàrquing, des d'un escriptori remot, sense necessitat d'estar present en el lloc.

7 Aspectes ambientals

En aquest estudi no s'esmenten aspectes ambientals, ja que es basa en el desenvolupament d'un algoritme, el qual no tindrà cap repercussió per al medi ambient.

8 Aspectes de seguretat

En aquest estudi no s'esmenten aspectes de seguretat, ja que es basa en el desenvolupament d'un algoritme, el qual no afectarà la seguretat de cap individu.

9 Planificació de la següent fase

L'algoritme desenvolupat i explicat al cinquè punt del treball forma part del prototip de la primera fase per a crear un programa informàtic que gestioni l'accessibilitat d'un pàrquing exterior.
Es tracta d'un prototip ja que per a donar una funcionalitat perfecte a aquest algoritme, ha de ser capaç de determinar les coordenades de cada vehicle estacionat (o eliminar les d'un vehicle que ha abandonat l'escenari) quan es mogui més d'un vehicle dins el pàrquing.

La segona fase de l'algoritme és, mitjançant les funcions existents a la llibreria OMPL (*Open Motion Planning Library*), desenvolupar un algoritme que simuli una planificació del moviment de cada vehicle i identifiqui si aquest pot sortir del pàrquing i identifiqui els possibles obstacles que hi puguin haver (els quals seran els altres vehicles, si aquests es troben mal estacionats). Aquesta planificació es basa en, donades les mides i posició d'un vehicle, els obstacles de l'escenari i una coordenada com a punt final (sortida del pàrquing), simular tots els moviments i rutes que pot efectuar el vehicle fins a poder o no arribar a la sortida (Rice University, 2014).

Finalment caldrà mostrar si hi ha algun vehicle que impedeix el moviment o sortida d'un altre per pantalla, podent així, avisar immediatament al conductor de què el seu vehicle esta mal aparcat.

Per tant, les tasques a realitzar en un futur amb l'objectiu de crear un programa informàtic real que gestioni l'accessibilitat d'un pàrquing són:

1. Afegir modificacions a l'algoritme actual per a permetre la identificació i coordenades d'estacionament de cada vehicle per separat que es mogui simultàniament dins l'escenari. Caldria utilitzar funcions complexes d'etiquetatge (o *labeling*), per a poder diferenciar cada vehicle.
2. Traduir tota la primera fase de l'algoritme al llenguatge de programació C++. A l'hora de compilar el programa, el llenguatge C++ permet una execució molt més ràpida de cada línia en comparació amb Python. Com que el programa final inclourà operacions complexes de visió artificial i simulació de moviment, caldrà una gran potència de processament. Aquesta millora en la capacitat de processament de l'algoritme és essencial en el cas d'utilitzar el Raspberry Pi ja que aquest no disposa de tanta potència per a gestionar informació com, per exemple, un ordinador de sobretaula.

3. Investigar i definir quines funcions de la llibreria OMPL caldrà utilitzar per a poder crear una simulació del moviment d'un vehicle dins un escenari sota uns obstacles definits.
4. Integrar les funcions que s'utilitzaran de la llibreria OMPL dins l'algoritme global donant com a entrada les coordenades dels requadres virtuals que encabiran cada vehicle estacionat, aconseguint així realitzar un *motion Planning* o simulació del moviment. Aquestes seran conegudes gràcies als resultats de la primera part de l'algoritme.
5. Desenvolupar un entorn gràfic que encabeixi tot l'algoritme per a convertir-lo en un programa informàtic. Aquest programa mostrarà per pantalla quin vehicle estacionat impedeix el moviment o sortida d'un altre vehicle.
6. Provar la funcionalitat del programa sota l'entorn del Raspberry Pi en pàrquings reals tot podent calibrar certs aspectes d'aquest per a adequar-lo a cada situació.

S'ha realitzat una planificació temporal fent una estimació de la duració de desenvolupament de cada tasca, fins a obtenir el programa informàtic totalment funcional. Aquesta planificació començaria la primera setmana de setembre acabant aproximadament a mitjans del mes d'abril de 2016.

Data d'inici	Codi tasca	Identificació	Precedida per	Esforç (dies)
1/09/2015	1	Modificar algoritme	-	41
12/10/2015	2	Traduir algoritme a C++	1	28
09/11/2015	3	Investigar eines OMPL	2	35
14/12/2015	4	Motion planning	3	35
04/01/2016	5	Desenvolupar entorn gràfic	4	57
01/03/2016	6	Proves finals de funcionalitat	6	45

Figura 37: Planificació de la continuació de l'estudi

10 Conclusions

Aquest estudi ha servit per a desenvolupar un prototip de la primera fase d'un programa informàtic que serviria per a poder gestionar un pàrquing de forma remota. El programa podria indicar en el precís moment en què un vehicle ha estacionat, si la posició d'aquest podria impedir la sortida del pàrquing d'algun dels altres vehicles ja estacionats. Com a conseqüència, es podria avisar al conductor que hauria de moure el vehicle a una nova posició, evitant possibles futurs col·lapses en temps real.

El prototip d'aquesta primera fase serveix per a concloure que és possible seguir el moviment d'un vehicle en un pàrquing utilitzant únicament funcions de visió artificial d'una llibreria gratuïta, així com demostrar que els canvis de lluminositat a l'escenari (excloent un escenari nocturn o amb molt baixa lluminositat) no provocarien una greu confusió dins l'algoritme gràcies a l'ús de l'eina **cv2.BackgroundSubtractorMOG** i al filtre d'àrees per mides.

Cal afegir que un cop haver desenvolupat el prototip per a què pugui identificar un vehicle, el següent pas seria poder separar el seguiment de cada vehicle que entrés o surtis simultàniament de l'escenari utilitzant funcions d'etiquetatge. Però aquesta modificació en l'algoritme es faria partint del sistema d'anàlisi ja desenvolupat en aquest estudi.

A la segona fase per al desenvolupament del programa, la simulació del moviment dels vehicles, no hauria d'aparèixer problemes ocasionats pel sistema d'anàlisi desenvolupat, ja que gràcies a la posició i requadre que encabeix cada vehicle s'utilitzaria com a possible obstacle dins l'escenari. Per tant, simulant tots els moviments i camins possibles d'un vehicle per aconseguir arribar a la sortida del pàrquing es podria identificar quin o quins vehicles són els que impedeixen aquesta sortida.

Finalment, la conclusió de l'estudi és que és possible crear un programa d'enorme utilitat per a gestionar l'accessibilitat en tota mena de pàrquings exteriors. Fins i tot en pàrquings creats per a grans esdeveniments culturals on l'enorme flux de vehicles i la nul·la definició de les places d'estacionament ocasiona importants col·lapses.

11 Bibliografia

- Bartolomé Sintes, M. (28 / Març / 2014). Recollit de http://www.mclibre.org/consultar/python/lecciones/python_while.html
- Bartolomé Sintes, M. (27 / Gener / 2015). Recollit de http://www.mclibre.org/consultar/python/lecciones/python_for.html
- Bartolomé Sintes, M. (4 / Febrer / 2015). Recollit de http://www.mclibre.org/consultar/python/lecciones/python_if_else.html
- Binario, C. (2012). Recollit de http://binarycodefree.blogspot.com.es/2010/01/teclas-del-teclado-y-valores-de-codigos.html
- Bradski, G. (2008). *OpenCV. Computer software.* Recollit de OpenCV Dev Zone: http://mirror.vicos.si/ros/wiki/attachments/Events(2f)ICRA2010Tutorial/ICRA_2010_OpenCV_Tutorial.pdf
- Foundation Raspberry Pi. (2008). Recollit de https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md
- Foundation Raspberry Pi. (2008). Recollit de https://www.raspberrypi.org/documentation/hardware/camera.md
- González Duque, R. (2010). *Python para todos.* Recollit de dspace.universia.net: http://dspace.universia.net/bitstream/2024/919/1/Python_para_todos.pdf
- Luciano, A. B. (Abril / 2013). *Wordpress*. Consultat el Març / 2015, a Visartblog: https://visartblog.wordpress.com/2013/05/02/resumen-historia-de-la-vision-artificial/
- Mordvintsev, A., & Abid, K. (31 / Octubre / 2014). Recollit de http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_gui/py_video_display/py_video_display.html
- Mordvintsev, A., & Abid, K. (31 / Octubre / 2014). Recollit de http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_video/py_bg_subtraction/py_bg_subtraction.html
- Mordvintsev, A., & Abid, K. (31 / Octubre / 2014). Recollit de http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_imgproc/py_contours/py_contour_features/py_contour_features.html
- Oliphant, T. E. (2006). *A guide to NumPy (Vol. 1, p. 85).*

- OpenCV dev team. (25 / Febrer / 2015). Recollit de http://docs.opencv.org/modules/highgui/doc/user_interface.html#namedwindow
- OpenCV dev team. (25 / Febrer / 2015). Recollit de http://docs.opencv.org/modules/imgproc/doc/geometric_transformations.html#getperspectivetransform
- OpenCV dev team. (25 / Febrer / 2015). Recollit de http://docs.opencv.org/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html#findcontours
- OpenCV, L. l. (2012). Consultat el Març / 2015, a Arévalo, V.M.: http://mapir.isa.uma.es/varevalo/drafts/arevalo2004lva1.pdf
- Python Foundation, S. (10 / Maig / 2015). Recollit de https://docs.python.org/2/library/time.html
- Python Foundation, S. (10 / Maig / 2015). Recollit de https://docs.python.org/2/tutorial/datastructures.html
- Python Foundation, S. (23 / Maig / 2015). Recollit de https://docs.python.org/2/reference/simple_stmts.html#the-return-statement
- Python Foundation, S. (23 / Febrer / 2015). Recollit de https://docs.python.org/2/library/functions.html?highlight=apply#apply
- Python Foundation, S. (10 / Maig / 2015). Recollit de https://docs.python.org/2/tutorial/inputoutput.html
- Rice University. (26 / Octubre / 2014). Recollit de http://ompl.kavrakilab.org
- Shermal, F. (2013). Recollit de http://opencv-srf.blogspot.com.es/2011/11/mouse-events.html
- Upton, E., & Halfacree, G. (2012). *Meet the Raspberry Pi.*
- Utkarsh. (Març / 2012). *OpenCV vs Matlab*. Recollit de blog.fixiatonal: http://blog.fixational.com/post/19177752599/opencv-vs-matlab